**JICSUM/D**
**Revision 1**
January 1996

# M68HC705J1A

# IN-CIRCUIT SIMULATOR

# USER'S MANUAL

© MOTOROLA, Inc., 1996; All Rights Reserved.

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.

Motorola and the Motorola logo are registered trademarks of Motorola Inc.

Motorola Inc. is an Equal Opportunity/Affirmative Action Employer.

IBM is a registered trademark of IBM Corporation.

ICS05J1A, RAPID, and CASM5J1A are © P&E Microcomputer Systems, Inc.,* 1994, 1995; All Rights Reserved. Portions of the software are © Borland International, 1987. Portions of the software are © Turbopower Software, 1988.

* P&E Microcomputer Systems, Inc.,
PO Box 2044
Woburn, MA 01888-2044
(617)-944-7585

# CONTENTS

## CHAPTER 1 OVERVIEW

1.1 Features and Specifications ..... 1-1
1.2 Quick Start ..... 1-2
1.3 Kit Hardware ..... 1-3
1.4 Programming Quick Start ..... 1-7
1.5 Simulation Speed ..... 1-8

## CHAPTER 2 OPERATING THE SIMULATOR

2.1 Starting ICS05J1A Software ..... 2-1
2.2 Main Screen ..... 2-2
2.2.1 CPU Window ..... 2-3
2.2.2 Code Window ..... 2-3
2.2.3 Variables Window ..... 2-5
2.2.4 Memory Window ..... 2-6
2.2.5 Debug Window ..... 2-6
2.2.6 Secondary Windows ..... 2-7
2.3 Hotkeys ..... 2-7
2.4 Help System ..... 2-9
2.5 Numbers ..... 2-9
2.6 Colors ..... 2-10
2.7 Script Files ..... 2-10
2.8 Log, Capture, and History Information ..... 2-10
2.9 Error Messages and Warnings ..... 2-11

## CHAPTER 3 COMMANDS

3.1 Command Syntax ..... 3-1
3.2 ICS05J1A Commands ..... 3-1
A Set Accumulator Value ..... 3-6
ACC Set Accumulator Value ..... 3-7
ASM Assemble Instructions ..... 3-8
BELL Sound Bell ..... 3-9
BF Block Fill ..... 3-10
BR Set Instruction Breakpoint ..... 3-11
BREAKA Set Accumulator Breakpoint ..... 3-12
BREAKSP Set Stack Pointer Breakpoint ..... 3-13
BREAKX Set Index Breakpoint ..... 3-14
C Set/Clear C Bit ..... 3-15
CAPTURE Capture Changed Data ..... 3-16
CAPTUREFILE Open Capture File ..... 3-17

3.2 ICS05J1A Commands (continued):


CCR Set Condition Code Register ..... 3-18
CF Open Capture File ..... 3-19
CLEARMAP Clear Map File ..... 3-20
CLEARSYMBOL Clear User Symbols ..... 3-21
COLORS Set Screen Colors ..... 3-22
COUNTER Set Instruction Counter ..... 3-23
CY Set Cycles Counter ..... 3-24
CYCLES Set Cycles Counter ..... 3-25
DASM Disassemble Memory ..... 3-26
DDRA Set Port A Direction Register ..... 3-27
DDRB Set Port B Direction Register ..... 3-28
DUMP Dump Memory to Screen ..... 3-29
EVAL Evaluate Expression ..... 3-30
EXIT Exit Program ..... 3-31
G Begin Program Execution ..... 3-32
GO Begin Program Execution ..... 3-33
GOMACRO Execute Macro after Break ..... 3-34
GOTIL Execute until Location ..... 3-35
GOTOCYCLE Execute until Cycle Counter Value ..... 3-36
H Set/Clear H Bit ..... 3-37
HELP Open Help System ..... 3-38
HIDECHIP Close Chip Window ..... 3-39
HISTORY History Capture ..... 3-40
HISTORYLOG Log History Capture ..... 3-41
I Set/Clear I Bit ..... 3-42
INFO Display Line Information ..... 3-43
INPUTA Set Port A Inputs ..... 3-44
INPUTB Set Port B Inputs ..... 3-45
INPUTS Show Port Inputs ..... 3-46
INT Set IRQ Pin State ..... 3-47
IRQ Set IRQ Pin State ..... 3-48
LF Open/Close Log File ..... 3-49
LISTOFF Do Not Show Steps ..... 3-50
LISTON Show Steps ..... 3-51
LOAD Load S19 File ..... 3-52
LOADMAP Load Map File ..... 3-53
LOGFILE Open/Close Log File ..... 3-54
MACRO Execute Script File ..... 3-55
MACROEND Stop Saving Commands ..... 3-56
MACROSTART Save Debug Commands ..... 3-57
MD Display Memory ..... 3-58
MEM Modify Memory ..... 3-59
MM Modify Memory ..... 3-60

3.2 ICS05J1A Commands (continued):


N Set/Clear N Bit ..... 3-61
NOBR Remove Breakpoints ..... 3-62
NOCNT Remove Instruction Counters ..... 3-63
PC Set PC Value ..... 3-64
POD Check Serial Port ..... 3-65
PORTA Set Port A Output Latches ..... 3-66
PORTB Set Port B Output Latches ..... 3-67
PROGRAM Program EPROM ..... 3-68
PRTA Set Port A Output Latches ..... 3-69
PRTB Set Port B Output Latches ..... 3-70
QUIET Suspend Window Updates ..... 3-71
QUIT Exit Program ..... 3-72
R Set Up System Registers ..... 3-73
REG Show Registers ..... 3-74
REM Add Comment to Script File ..... 3-75
RESET Simulate Processor Reset ..... 3-76
RESET_COUNTS Reset Instruction Counters ..... 3-77
RESETGO Reset and Restart MCU ..... 3-78
RUN Begin Program Execution ..... 3-79
SCRIPT Execute Script File ..... 3-80
SHELL Shell to DOS ..... 3-81
SHOW Display Memory ..... 3-82
SHOWCHIP Open Chip Window ..... 3-83
SHOWCODE Display Code ..... 3-84
SHOWPC Display Code at PC ..... 3-85
SNAPSHOT Save Screen ..... 3-86
SOURCE Toggle Code Display ..... 3-87
SP Set SP Value ..... 3-88
SS Source Step ..... 3-89
ST Single Step (Trace) ..... 3-90
STACK Display Stack ..... 3-91
STEP Single Step (Trace) ..... 3-92
STEPFOR Step Forever ..... 3-93
STEPTIL Step Until Location ..... 3-94
SYMBOL Add Symbol ..... 3-95
SYSINFO Show System Information ..... 3-96
T Single Step (Trace) ..... 3-97
UPLOAD_SREC Upload S-Record ..... 3-98
VAR Display Variable ..... 3-99
VER Display Version ..... 3-100
VERSION Display Version ..... 3-101
WAIT Wait N Cycles ..... 3-102
WHEREIS Display Symbol Value ..... 3-103

3.2 ICS05J1A Commands (continued):
X Set X Register Value ..... 3-104
XREG Set X Register Value ..... 3-105
Z Set/Clear Z Bit ..... 3-106
ZOOM Resize Source Window ..... 3-107

**CHAPTER 4 SUPPORT INFORMATION**

4.1 Functional Description.....4-1
4.1.1 Emulation .....4-1
4.1.2 Programming .....4-2
4.2 Quick-Start Troubleshooting Guide.....4-2
4.3 Programming Troubleshooting Guide.....4-4
4.4 Logic Diagram.....4-4

**Index** ..... index-1

## FIGURES

**Figure** **Page**

1-1 M68HC705JICS Circuit Board.....1-4
2-1 ICS05J1A Main Screen.....2-2
4-1 JICS Logic Diagram.....4-5

## TABLES

**Table** **Page**

1-1 M68HC705JICS Specifications .....1-2
2-1 ICS05J1A Hotkeys .....2-8
3-1 ICS05J1A Commands .....3-2

# CHAPTER 1
# OVERVIEW

The M68HC705J1A In-Circuit Simulator Kit (M68HC705JICS) consists of a small printed circuit board, ICS05J1A simulator software, the CASM5J1A assembler, and the RAPID integrated development environment. The kit works with any PC (or clone) host computer that has 640 kilobytes of memory (either floppy-based or hard disk) and has a standard serial port.

The board and the ICS05J1A software together form a full, non-realtime simulator and I/O emulator for the MC68HC705J1A microcontroller unit (MCU). This simulator lets you debug your code in an easy-to-use, software-based environment. Furthermore, you may connect the board to your target hardware, to use actual inputs and outputs of your target during simulation of your code. After debugging is done, you can use the same system to program your code into MC68HC705J1A MCUs.

Chapter 2 explains ICS05J1A operation, identifying screen displays and windows. Chapter 3 explains ICS05J1A commands. Chapter 4 gives support information: functional description, troubleshooting guide, and logic diagram.

For simulator quick-start instructions, see paragraph 1.2. Paragraph 1.3 gives M68HC705JICS hardware information. Paragraph 1.4 gives quick-start instructions for programming blank 705J1A MCUs. Paragraph 1.5 gives an exercise for approximating the speed of your simulation.

The kit software diskette includes RAPID, an integrated development environment from P&E Microcomputer Systems. This version of RAPID is configured for the CASM5J1A assembler and the ICS05J1A simulator. The simulator kit includes a RAPID user's manual, with separate sheets that explain the enhancements for CASM5J1A and ICS05J1A.

## 1.1 FEATURES AND SPECIFICATIONS

M68HC705JICS kit features include:

- An economical means of developing and debugging target systems incorporating MC68HC705J1A MCUs.
- RAPID integrated development environment that combines program assembly, editing, and in-circuit simulation.
- Four types of software breakpoints.
- Communication with the host computer via any of eight serial ports.
- DB9 serial communication cable and 9-to-25-pin adapter.

- 20-lead emulation cable, for connection to your target system.
- Ability to program MC68HC705J1A EPROM MCUs.
- Wall-plug power supply, for running the MC68HC705JICS from 120-volt line power.

Table 1-1 lists M68HC705JICS specifications.

**Table 1-1. M68HC705JICS Specifications**

| Characteristic | Specifications |
|---|---|
| Temperature | |
| Operating | Room temperature ± 10° C |
| Storage | -40° to +85° C |
| Relative humidity | 0 to 95% (non-condensing) |
| Power requirements | +9 Vdc @ 0.2 A (from transformer that inputs 120-volt ac line power ) |
| Dimensions | 3.6 x 6.1 in. (91 x 154 mm) |

## 1.2 QUICK START

Follow the numbered steps below for a quick-start, hands-on introduction to the M68HC705JICS kit.

1. Install the JICS programs from the distribution diskette: insert the diskette into the appropriate drive, then type **`A:install`** or **`B:install`**, according to the drive letter. Follow the instructions that appear on the screen.
2. Use the provided DB9 cable to connect the M68HC705JICS circuit board to the COM1 serial port of you computer. If necessary, use the 9-to-25-pin adapter between the cable and the computer serial port. Plug the other end of the cable into circuit-board connector P2.
3. Plug the transformer into a wall socket; plug the power cable into circuit-board connector P1. Turn switch S1 ON to apply power to the circuit board.
4. At the DOS prompt, type **`RAPID PODTEST`** then press the **`Enter`** key; the RAPID editor display appears.
5. Press the **`F4`** key to assemble the `PODTEST` program.
6. Press the **`F6`** key to go directly to the ICS05J1A simulator. (Should the ICS05J1A main screen not appear, check that file `ICS05J1A.EXE` is in the current directory.)

7. Type **GO** to start code simulation; LED1 begins to flash. (Should LED1 not flash, make sure that the fabricated jumper is inserted correctly in jumper header J2. If it is and LED1 still does not flash, consult the quick-start troubleshooting guide of Chapter 4.)
8. Press and hold circuit-board switch S4 one or more times. Each time you do, LED1 stops flashing. (If LED1 does not stop flashing when you press S4, make sure that the fabricated jumper is inserted correctly in header J1. If the jumper is installed correctly, but LED1 continues to flash, consult the quick-start troubleshooting guide of Chapter 4. Alternatively, call Global Data Specialists (800-451-3464 or 602-437-4331) for a replacement.)
9. To stop program execution, press any key.
10. To return to the assembler, type **EXIT**.
11. To leave RAPID and return to DOS, press **F5**. This completes the quick start.

## 1.3 KIT HARDWARE

Figure 1-1 shows the M68HC705JICS kit hardware: a single printed circuit board. Connect the board to your host computer via the DB9 serial cable; run the cable between the computer serial port and board connector P2. (If the host computer serial port is a 25-pin connector, use the 9-to25-pin adapter between the port and the cable.) Note that if you do not connect the circuit board to the computer, the ICS05J1A software works as a full standalone simulator.

Connector P1 is for 9-volt operating power for the board. Plug the output connector of the 120Vac-to-9Vdc transformer into this connector.

Connector P2 is for communication with the host computer. This RS232 connector is for the DB9 cable that runs to the computer.

To connect to your target system, run the factory-supplied 20-lead header cable between board socket U5 and the 20-pin DIP socket of the target system. The small, boxed number 1 in Figure 1-1 identifies pin 1 of socket U5. (When you connect the JICS board to your target system and power up, the target system provides all inputs to the ICS05J1A software, and accepts all outputs from the ICS05J1A software. `RESET` and `IRQ` lines also interact with the ICS05J1A software. Note that when you execute a **GO** command, the simulator does not run in real time. But the faster your host computer, the faster the simulation.)

Switch S1 is the board power switch. Switch S2 controls programming (VPP) power. Switch S3 is the reset switch for the microprocessor at board location U4. Switch S4 is for user experimentation with the PA0 line.

**NOTE**

Do *not* press switch S3 unless the computer displays a message to do so.

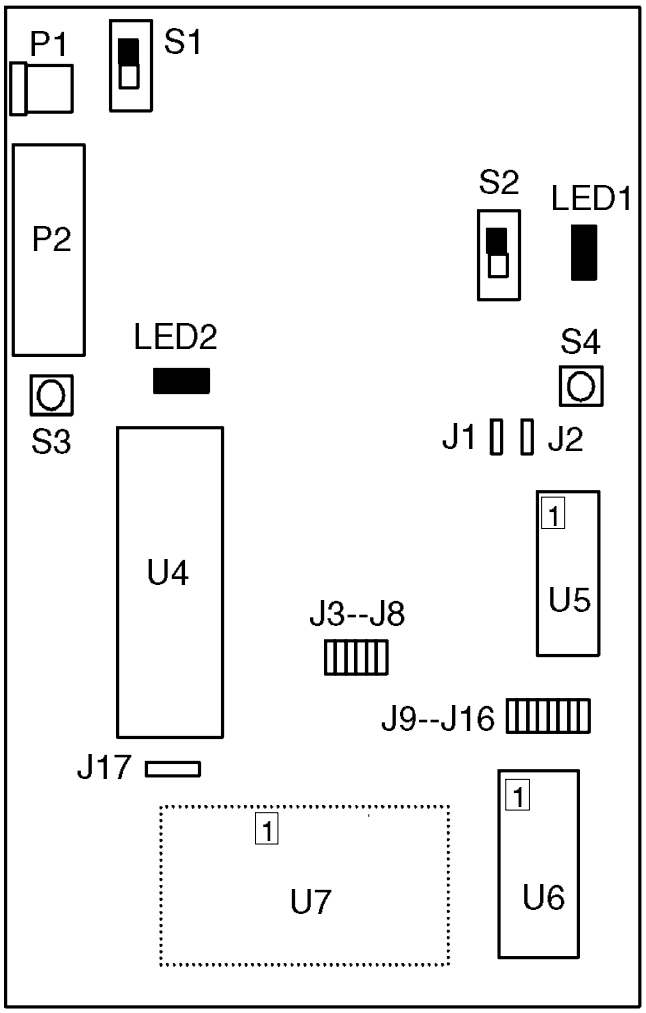
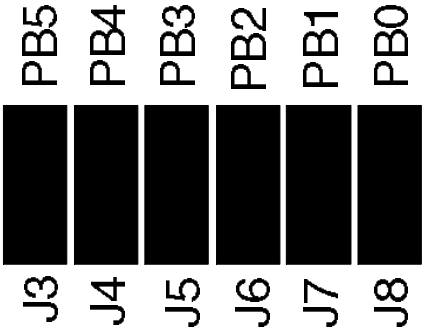



**Figure 1-1. M68HC705JICS Circuit Board**

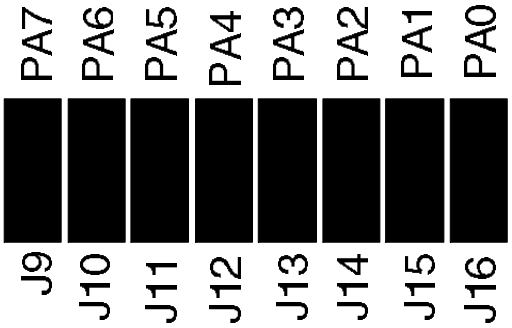
Jumper header J1 is the switch S4 enable: a fabricated jumper in header J1 connects switch S4 to the PA0 pin of the MCU. To disconnect switch S4, remove the jumper from this header.

Jumper header J2 is the LED1 enable: a fabricated jumper in this header connects LED1 to the PA7 pin of the MCU. To disconnect LED1, remove the jumper from this header.

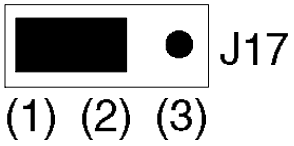
Jumper headers J3 through J8 are compensating pull-downs for the port B pins, as the diagram below shows. The M68HC705JICS kit uses an MC68HSC705C8A MCU to emulate an 68HC705J1A MCU. But unlike the MC68HC705J1A, the MC68HSC705C8A does not have programmable pull-downs. Jumper headers J3 through J8 compensate manually: a fabricated jumper in any of these headers connects the corresponding port pin to a pull-down resistor, pulling the signal low.




Jumper headers J9 through J16 are compensating pull-downs for the port A pins, as the diagram below shows. Fabricated jumpers in any of these headers have the same function as jumpers in headers J3 through J8, but apply to port A pins instead of port B pins.



Jumper header J17 sets the baud rate for communication with the host computer. The diagram below shows the factory configuration: the fabricated jumper between pins 1 and 2 selects 115.2 kilobaud.




For the alternative rate of 19.2 kilobaud:

- Position the J17 jumper between pins 2 and 3.
- Reset the JICS circuit board, by pressing the S3 switch.
- Include the /b19200 parameter value in the ICS05J1A startup command. (Paragraph 2.1 explains more about this parameter value.)

## 1.4 PROGRAMMING QUICK-START

Sockets U6 and U7 are for programming MC68HC705J1A MCUs: U6 for 20-pin DIP MCUs, U7 for 20-pin SOIC MCUs. Follow the numbered steps below to program an MCU.

**NOTES**

This quick-start does program a J1A MCU. If you follow the steps below, you must erase the MCU before using it again.

Socket U7 accomodates SOIC MCU packages of as many as 28 pins. To use this socket for an SOIC J1A MCU, which has only 20 pins, unlatch and open the lid, insert the MCU pins into the 20 socket holes nearest jumper header J17, then close and latch the socket lid.

1. Make sure that no MCUs are in either socket U6 or U7.
2. Calibrate VPP. To do so, turn power switch S1 ON, then turn VPP switch S2 ON. Measure the signal on pin 19 of socket U6 or U7. If necessary, adjust potentiometer VR1 until your meter reads 16.5 volts. Turn switches S2 and S1 OFF. (This completes VPP calibration. )
3. From within ICS05J1A, type **`PROGRAM`**. Follow the programming instructions that appear on the screen, then select `BLANK CHECK`.
4. If the MCU is blank, select `PB` to program the EPROM and the MOR from the program previously loaded in the simulator. Other options include selecting `PE` (to program only the EPROM) and `PM` (to program only the MOR register).
5. Check the functionality of programming:
   a. Select `QU` to quit the programmer, then follow the screen instructions to exit both ICS05J1A and RAPID.
   b. Make sure that switch S1 (VCC) is ON; make sure that switch S2 (VPP) is OFF. Then restart ICS05J1A without loading a program.
   c. Type **`PROGRAM`**, then follow the screen instructions.
   d. Select `SU` to specify a .S19 upload file; enter filename **`TEST.S19`**.
   e. Select `UL` to upload the EPROM to the .S19 file.
   f. Select `QU` to quit the programmer and return to ICS05J1A; follow the screen instructions. Make sure that switch S1 (VCC) is ON; make sure that switch S2 (VPP) is OFF.
   g. Type **`LOAD TEST`** to load the .S19 file into the simulator.
   h. Type **`PC 300`** to set the program counter value to *300*.
   i. Type **`GO`** to run the program; verify that LED1 flashes.

Should programming not be successful (for example, if the MCU does not verify), consult the programming trouble-shooting guide in Chapter 4.

## 1.5 SIMULATION SPEED

Your host computer simulates execution of instructions; this simulation is noticeably slower than an MCU's actual execution of the instructions. Furthermore, this simulation takes more time for certain instructions than for others, preventing a straightforward calculation of simulation speed. But you can get a feel for the simulation speed by calculating the *actual* speed; to do so, follow the steps below.

**NOTE**

An *actual* speed of 10 kilohertz means that the simulator on your computer is running at the same speed as the real J1A MCU with a 20-kilohertz crystal (a divide-by-2 is attached to the internal oscillator output). Typical values for the *actual* speed are 3 to 50 kilohertz.

1. Use the **`LOAD`** command to load your code.
2. Set the PC value to the beginning of your routine.
3. Clear the cycle counter by typing **`CY 0`**, then pressing **`Enter`**.

**NOTE**

Entering the **`QUIET`** command at this point will increase execution speed, by suspending updates to main-screen windows. Chapter 3 gives additional information about the **`QUIET`** command.

4. Get your stopwatch ready.
5. Type **`G`**. Then simultaneously start the stopwatch and press **`Enter`**; execution begins.
6. After 10 seconds, simultaneously stop the watch and press the space bar of your keyboard; execution stops.
7. Type **`CY Enter`**; the cycle count appears, in hexadecimal and decimal formats.
8. Divide the decimal cycle count by 10. The result is your actual speed, in hertz.

# CHAPTER 2
# OPERATING THE SIMULATOR

This chapter explains how to start and use the ICS05J1A simulator and I/O emulator. ICS05J1A is a full MC68HC705J1A software simulator, which uses these types of files:

- **Object files** — files in .S19 format, produced by CASM5J1A and other assemblers. These files contain the actual code that is simulated. You may specify such a file on the command line or load it via the **`LOAD`** command.
- **Map files** — files in map-file format, which allow symbolic and source-level debugging. You may specify such a file on the command line or load it via the **`LOADMAP`** command.
- **Script files** — plain ASCII files containing ICS05J1A commands. Paragraph 2.8 gives more information about script files.

## 2.1 STARTING ICS05J1A SOFTWARE

To get into the simulator directly, follow the instructions of this paragraph. (To get into the simulator indirectly, through the RAPID integrated development environment, follow the instructions in the RAPID user's manual.)

At the DOS prompt ( > ) type:

**`ICS05J1A [bw] [`*`filename`*`] [`*`n`*`] [/b19200 | /b115200] [*]`**

where:

bw — Optional parameter value for a black and white display.

*filename* — Optional name of .S19 and .map files the simulator is to load.

*n* — Optional serial port number (1 is the default).

/b19200 — Optional parameter value for communications at 19200 baud. (For this parameter value, you also must reconfigure jumper header J17 and reset the JICS board, per paragraph 1.3.)

/b115200 — Optional parameter value for communications at 115200 baud (the default).

* — Optional parameter value for by-passing the opening screen.

Spaces must separate parameter values. You may enter parameter values in any order.

Examples of startup commands are:

| | |
|---|---|
| **>ICS05J1A bw** | Start simulator, use black-and-white display. |
| **>ICS05J1A myprog *** | Start simulator, by-passing the opening screen, load files MYPROG.S19 and MYPROG.MAP, then simulate a reset. |
| **>ICS05J1A 2 myprog** | Start simulator as for the previous example, but use serial port COM2. |
| **>ICS05J1A /b19200** | Start simulator, communicate at 19200 baud. (The J17 jumper must be between pins 2 and 3, as paragraph 1.3 explains.) |

Immediately upon startup, the ICS05J1A runs the script file STARTUP.05J, if such a file is in the current directory. Paragraph 2.7 explains more about script files.

The ICS05J1A main screen appears, to begin your simulator session.

## 2.2 Main Screen

Figure 2-1 shows the ICS05J1A main screen, which consists of five primary windows: CPU, code, variables, memory, and debug. Paragraphs 2.2.1 through 2.2.5 explain these windows.

Centered among the primary windows are two lines of the form POD: yes and HC705J1A. The first line confirms communication between the JICS circuit board and the host computer; the second line identifies the HC05 MCU being simulated.

ICS05J1A also includes several temporary windows, which appear superimposed over the main screen when needed. Paragraph 2.2.6 explains temporary windows.

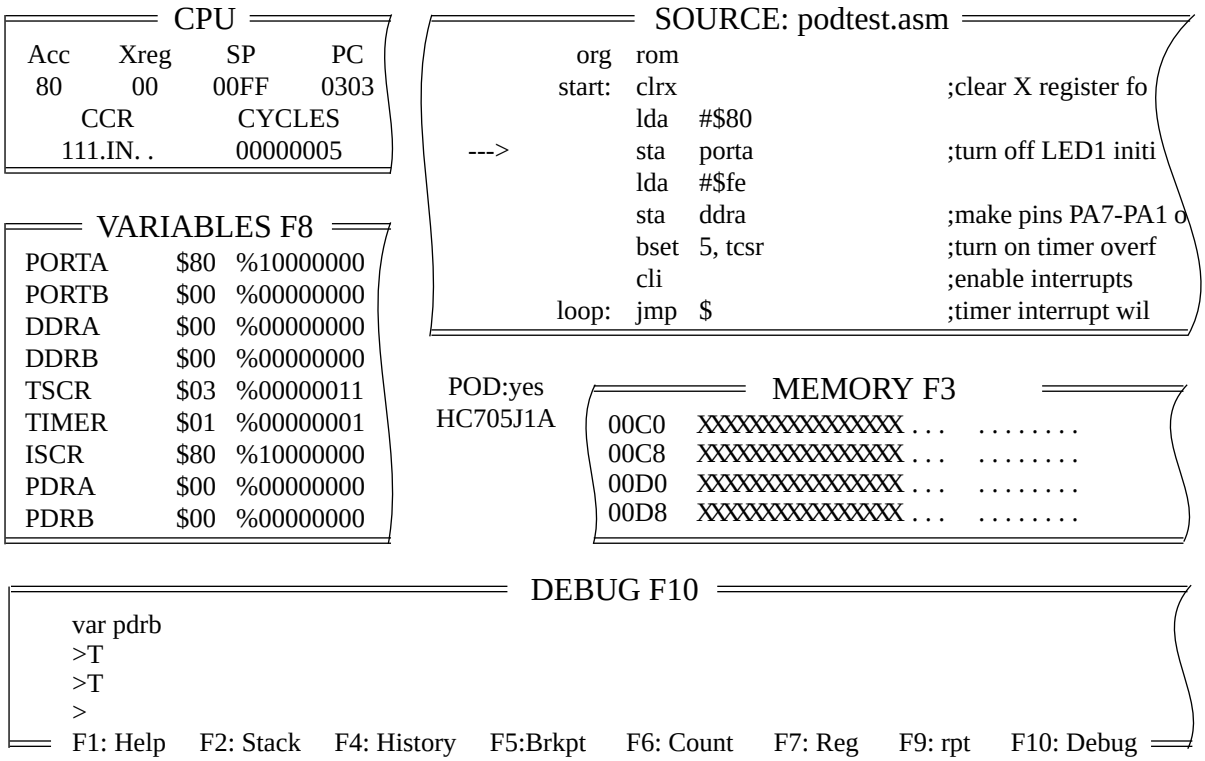



**Figure 2-1. ICS05J1A Main Screen**

### 2.2.1 CPU Window

```
CPU
Acc    Xreg    SP     PC
 XX      XX   00FF   0300
    CCR         CYCLES
  111.I...   00000000
```

The CPU window is at the upper left of the main screen. This window shows the current values in the accumulator, X (index) register, stack pointer (SP), program counter (PC), condition code register (CCR), and cycle counter.

The CCR flag pattern is `111HINZC`: H is half-carry, I is IRQ interrupt mask, N is negative, Z is zero, and C is carry. A letter in the CCR value indicates that the corresponding bit is set; a period means that the corresponding bit is clear. (The top three bits are not implemented; they always are set.)

For example, the CCR value `111.I..C` means that the interrupt mask and carry flags are set (have the value 1), but the other flags are clear (have the value 0).

The cycle counter shows the current cycle count, which allows timing of routines and procedures for time-critical (real-time) applications. For example, to calculate the time necessary for a portion of code to execute, multiply the cycle count by the target system's unit of time per cycle.

The maximum internal value of the cycles counter is $FFFFFFFF. You can assign a specific value to the cycles counter via the **`CY`** or **`CYCLES`** command.

### 2.2.2 Code Window

```
SOURCE: j1aport.asm
   start
-->        blcr 7,ICSR
           lda #$0f            ;write $0f to port A'
           sta ddra            ;and read it back on
           lda #$05            ;
           sta porta
           clra
           lda porta           ;store results at $e0
           sta pad1

           lda #$0a            ;write $0a to port A'
```

The code window is at the upper right of the main screen. This window shows your actual source code, provided that you have loaded a valid source-level .MAP file. The symbol --> shows where the program counter (PC) points in the code. The symbol <B> and red highlighting mark lines at which breakpoints are set. The symbol <C> and cyan highlighting mark lines at which instruction counters are set.

The grey highlight is the cursor for the code window. You may scroll through code in the window by using the arrow, **Home**, **End**, **Page Up**, and **Page Down** keys from within the debug window.

To set a breakpoint at the cursor line of code, press **Alt-B**. Press **Alt-P** to set the PC address value to the cursor line of code. To set an instruction counter at the cursor line of code, press **Alt-C**. Pressing **Alt-G** is like entering a **GOTIL** command, executing code from the current PC address value to the cursor line of code. To enlarge (zoom) the code window, press **Alt-Z**.

Always keep in mind that if you change your code via an **ASM**, **MEM**, or **MM** command, the source code in the code window no longer will be correct. (To make the code correct again, you must use RAPID and CASM5J1A to re-assemble the code.)

If no valid map file is loaded, or there is no source code at the current PC address value, the code window shows disassembled code at the current PC (highlighted in the second row). Following this line are the next eight instructions. The diagram below shows a disassembled-code display in the code window.

```
                              CODE
--->0300          9C        RSP
    0301          4F        CLRA
    0302          5F        CLRX
    0303          3FC1      CLR 00C1
    0305          A630      LDA #30
    0307          B708      STA TSCR
    0309          A600      LDA #0
    030B          B704      STA DDRA
    030D          A63F      LDA #3F
    030F          B701      STA PORTB
```

Column 1 of this display is the list of addresses (or source-code labels if you have loaded a .MAP file). Column 2 lists the corresponding hexadecimal object code. The first row shows the most recently-executed instruction. Any line that is the target of a breakpoint includes a commentabout this fact. The software truncates labels, if necessary, so that all the information fits into the window. You may scroll forward through the disassembled code, just as you can through source code.

**NOTE**

If the code window shows disassembled code, it is not possible to scroll backwards from the PC.

To toggle between source code and symbolic disassembly of the memory, use the **SOURCE** command. (A valid .MAP file must be loaded for the **SOURCE** command to work.)

The code window can display code in another module (created via a CASM5J1A INCLUDE directive). To see such code, press **Alt-M** and choose the appropriate module. Note that when ICS05J1A is simulating instructions, the code window always shows the module that contains code for the current program counter (PC) address value. You may press **Alt-B** to set a breakpoint in any module, not just the module that contains the current PC address value.

To search for a string in the current module, press **Alt-F**, enter the target string at the prompt, then press **Enter**. To find the next occurrence of the same target string, press **Alt-L**.

### 2.2.3 Variables Window

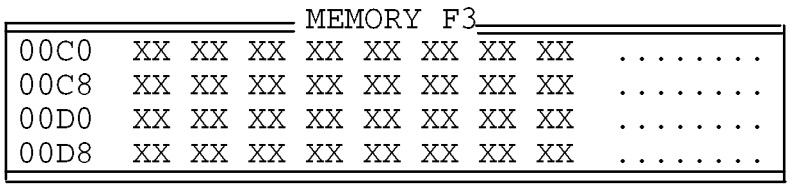
```
═══════ VARIABLES F8 ═══════
PORTA          $XX %XXXXXXXX
PORTB          $XX %XXXXXXXX
DDRA           $00 %00000000
DDRB           $00 %00000000
TSCR           $03 %00000011
TIMER          $00 %00000000
ISCR           $80 %10000000
PDRA           $00 %00000000
PDRB           $00 %00000000
```

The variables window is below the CPU window, at the left edge of the main screen. This window shows current values, in hexadecimal and binary formats, for specified byte, word, or string variables.

Use the **VAR** command to add a variable to the variables window. At ICS05J1A startup, the variables window shows values for these register variables:

| | |
|---|---|
| PORTA | port A data |
| PORTB | port B data |
| DDRA | port A direction register |
| DDRB | port B direction register |
| TSCR | timer station control register |
| TIMER | timer counter register |
| ISCR | interrupt status and control register |
| PDRA | port A pull-down register |
| PDRB | port B pull-down register |

### 2.2.4 Memory Window

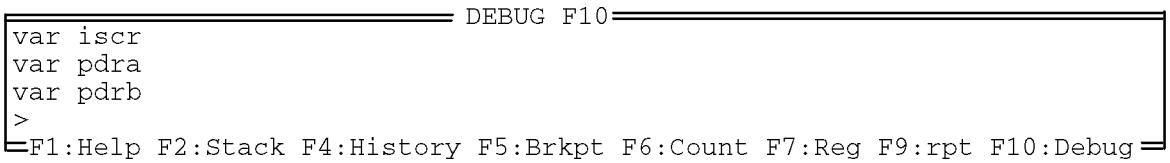



The memory window is below the code window, at the right edge of the main screen. This window shows current values of 32 consecutive memory locations, in hexadecimal and printable ASCII formats.

To enter this window, press the **F3** key; use arrow, **Page Up**, and **Page Down** keys for scrolling. When you are in the debug window, you may use the **MD** or **SHOW** command to set the window anywhere in memory.

You can edit values directly, when you are in the memory window. To do so, press the **F3** key: the grey highlight cursor appears at the first value. When you type a new value, the cursor automatically advances to the next value. To advance to the next value without changing the current value, press the → key.

### 2.2.5 Debug Window




The debug window extends along the bottom of the main screen. This is the default window: you are in this window automatically when the simulator starts. Enter all your commands in this window; error messages appear in this window. Hotkey definitions form the bottom of this window's frame. To return to this window from the code or memory windows, press the **F10** key.

### 2.2.6 Secondary Windows

Each secondary window appears only when needed, superimposed over the main screen. Making any of these windows disappear returns you to the debug window. The secondary windows include:

- **Breakpoint Window** — a display of all current breakpoints. To bring up this window, press **F5**. Press **F10** or **Esc** to make the breakpoint window disappear. (Entering the **BR** command without any parameter values shows a list of breakpoints in the debug window, but does not bring up the breakpoint window.)
- **Chip Visualization Window** — a depiction of the MCU itself, showing the role of each pin. Arrow symbols indicate whether each pin is programmed as an input or an output. A value outside an arrow symbol is the value currently being simulated on the pin. To bring up this window, enter the **SHOWCHIP** command. To make the chip visualization window disappear, enter the **HIDECHIP** command.
- **Color Selection Window** — a matrix of foreground/background color combinations for screen elements; you select colors for these elements from this window. To bring up this window, enter the **COLORS** command. Paragraph 2.6 and the Chapter 3 **COLORS**-command explanation give more information about screen colors.
- **History Window** — a full-screen display of history capture information: instructions, timer values, and CPU values during the time that history capture was enabled. To bring up this window, press **F4**. Press **F10** or **Esc** to make the history window disappear. Paragraph 2.8 and the Chapter 3 **HISTORY**-command explanation give more information about history capture.
- **Instruction Counter Window** — a display of instruction counters and values. To bring up this window, press **F6**. Press **F10** or **Esc** to make this window disappear. Chapter 3's explanations of the **COUNTER**, **NOCNT**, and **RESET_COUNTS** commands give more information about instruction counters. (Entering the **COUNTER** command without any parameter values shows a list of instruction counters in the debug window, but does not bring up the instruction counter window.)
- **Register Files Window** — a display of P&E register files, which list the purposes of MCU peripheral bits. (Register files are sold separately.) To bring up this window, press **F7** or enter the **R** command. Press **F10** or **Esc** to make this window disappear.
- **Stack Window** — a depiction of the ICS05J1A stack, the interrupt stack, and the subroutine stack. To bring up this window, type **F2**. Press **F10** or **Esc** to make the stack window disappear.

## 2.3 HOTKEYS

ICS05J1A includes several hotkeys: function keys, special-purpose keys, and Alt-key combinations that have specific uses. Table 2-1 lists these hotkeys. Note that several hotkeys are shortcut alternatives to entering ICS05J1A commands.

**Table 2-1. ICS05J1A Hotkeys**

| Hotkey | Function |
|---|---|
| F1 | Open HELP system. |
| F2 | Show STACK window. |
| F3 | Go to MEMORY window. |
| F4 | Go to HISTORY window. |
| F5 | Go to BREAKPOINT window. |
| F6 | Go to COUNT window. |
| F7 | Show REGISTER FILES window. |
| F8 | Go to VARIABLES window. |
| F9 | Repeat the last command entered in the debug window. |
| F10 | Go to DEBUG window |
| ↑ | Scroll up through code in window. |
| ↓ | Scroll down through code in window. |
| ← | Move cursor left through code in window. |
| → | Move cursor right through code in window. |
| Page Up | Page up through code in window. |
| Page Down | Page down through code in window. |
| Home | Go directly to start of code in window. |
| End | Go directly to end of code in window. |
| Alt-B | Set (or delete) breakpoint at current line in CODE window. |
| Alt-C | Set (or delete) counter at current line in CODE window. |
| Alt-F | Search for a string. |
| Alt-G | Execute code from PC address value to highlighted line. |
| Alt-L | Search for subsequent occurrence of a string. |
| Alt-M | Show a different module in CODE window. |
| Alt-P | Set PC address value to current line in CODE window. |
| Alt-Z | Zoom CODE window. |
| Shift-↑ (or Alt-U) | Duplicate most recently executed command (press **Enter** to execute). |
| Shift-↓ (or Alt-D) | Advance to the next command. |
| Esc | Back out of current position or return to the DEBUG window. |

## 2.4 HELP SYSTEM

The ICS05J1A features a pop-up help system. To access this system, press the **F1** key or enter the **HELP** command: the help menu appears. Position the cursor on a topic, then press **Enter** to bring up a page of information.

There are two ways to by-pass the help menu:

- Entering the topic as a parameter of the **HELP** command.
- Typing a topic at the command prompt in the debug window, but pressing **F1** instead of **Enter**.

A symbol at the bottom of help page indicates at least one more page of information. To move among multiple pages of help information, use the **Page Up** and **Page Down** keys. To back out of the help system, press **Esc** one or more times.

You may change the colors of the help screens via the **COLORS** command.

## 2.5 NUMBERS

You may change any values of the main screen from the debug window. To change a value in the memory window, use the **MEM** or **MM** command. To change code in the code window, use the **ASM** command. To change values in other windows, specify the name of the item and the new value.

The symbol XX indicates an uninitialized value. The symbol UU indicates simulation of a memory location unimplemented in the MCU.

The default number format for ICS05J1A is hexadecimal. To override this default, use the appropriate prefix: **!** for decimal, **@** for octal, or **%** for binary. You do not have to use the **$** prefix for hexadecimal values, but it is not an error to do so. (The ICS05J1A also recognizes optional format suffixes: **H** for hexadecimal, **T** for decimal, **O** for octal, and **Q** for binary. But you must *not* use both a prefix and a suffix for a value.)

```
$0FF = 255T = 0FFH = !255 = 11111111Q = %11111111
```

Examples:

| | |
|---|---|
| **>PC 100** | Change PC address value to $100. |
| **>N 1** | Assign value 1 to CCR N bit. |
| **>BR 205** | Set breakpoint at address $205. |
| **>BR END** | Set breakpoint at address of symbol END. |

## 2.6 COLORS

You may select colors for all elements of ICS05J1A displays. To do so, enter the **COLORS** command. This command brings up the color selection window: a matrix of foreground/background color combinations and a list of screen elements. Matrix rows correspond to background colors, matrix columns correspond to foreground colors; each color combination has a two-digit hexadecimal code.

When the window appears, the first screen element is highlighted. Work your way down the list of screen elements by entering the code for the color combination you want for the element, then pressing **Enter**. To retain the current color combination, press **Enter** without entering a new code. As the software accepts your entry, it highlights the next screen element. (To leave the color selection window without saving any new color combinations, press **Esc**.)

When you press **Enter** for the final screen element, the ICS05J1A saves the new set of colors in file `ICS05J.CFG`. As part of startup, the simulator looks for this file in its current directory. If it finds the file, the ICS05J1A uses the color choices of the file; otherwise, it uses default colors.

## 2.7 SCRIPT FILES

A script file is an ASCII text file of simulator commands. Although you can use most text editors to create script files, it is particularly easy to use RAPID for this purpose.

In a script file, each command must be on its own line. Script files can include any of the ICS05J1A commands.

**NOTE**

> Two ICS05J1A commands, **REM** and **WAIT**, are valid only in script files. The **REM** command tells the system to display a comment during source-file execution. The **WAIT** command delays execution of a subsequent script-file command. Chapter 3 includes more information about these two commands.

If a file named STARTUP.05J is in the same directory as ICS05J1A, the simulator treats this file as a script file, automatically running the file at startup.

## 2.8 LOG, CAPTURE, AND HISTORY INFORMATION

The ICS05J1A includes three ways to record simulation information for later reference:

- **Log files —** simple ASCII-text, scratch-pad files that the simulator creates, if you use the **LOGFILE** or **LF** command. A log file records entries of commands and copies of debug-window responses. You may view log files from within RAPID.
- **Capture files —** files the simulator creates, via the **CAPTUREFILE** or **CF** command, to record state or value changes at memory addresses you specify. (Use the **CAPTURE** command to specify addresses to be monitored for value changes.)

- **History buffer** — a circular buffer that captures instructions, timer values, and CPU values when you enable history capture (via the **HISTORY** command). If you also open a log file and use the **HISTORYLOG** command, ICS05J1A also puts the history information in the log file. You may open the history window (via the **F4** key) to view buffer contents directly.

## 2.9 ERROR MESSAGES AND WARNINGS

There are many possible error messages and warnings in ICS05J1A. Most of these messages are self-explanatory, referring to the instruction currently being simulated. A common form of error messages is:

```
>Attempt to use uninitialized (XX) memory.
```

In most cases, the message specifies an address or a CPU register. The probable error that caused the message is an uninitialized variable. Another likely cause for such an error message is trying to place an uninitialized register on the stack during interrupt processing. To get around this, enter a value into the A or X register. (It is good practice to always have known values in these registers.)

Remember that loading code into the simulator does *not* cause a CPU reset. Only a CPU reset (via the **RESET** command) makes the simulator load the reset vector into the program counter (PC).

# CHAPTER 3
# COMMANDS

Enter ICS05J1A commands at the bottom of the debug window. Table 3-1 is a list of these commands; explanations of each command follow the table.

## 3.1 COMMAND SYNTAX

Table 3-1 and the command explanations follow this syntax:

**COMMAND[.B|.W]** ***argument*** **[*argument* ...]**

where:

| | |
|---|---|
| COMMAND | A command name, which you must enter as shown (except that you may use lower-case letters). |
| .B or .W | A specific parameter value, which you must enter as shown (except that you may use lower-case letters). |
| *argument* | A representative word for an actual argument, in lower-case italic letters, such as *address*, *filename*, and *n*. For actual commands, you must use actual address, filename, or number values instead of such representative words. |
| [ ] | Brackets indicate optional items. |
| \| | A vertical line means *or*. |
| ... | An ellipsis means that you can repeat the preceding item. |

Except where otherwise noted, numerical values are hexadecimal. The command prompt ( > ), at the bottom of the debug window, indicates that the system is ready for another command.

## 3.2 ICS05J1A COMMANDS

Table 3-1 lists the commands; individual explanations follow the table.

**Table 3-1. ICS05J1A Commands**

| Syntax | Meaning |
|---|---|
| A *n* | Set accumulator value |
| ACC *n* | Set accumulator value |
| ASM *address* | Assemble instructions |
| BELL [*n*] | Sound bell |
| BF[.B\|.W] *startrange endrange n* | Block fill |
| BR [*address* [*n*]] | Set instruction breakpoint |
| BREAKA [*n* [*address*]] | Set accumulator breakpoint |
| BREAKSP [*n* [*address*]] | Set SP breakpoint |
| BREAKX [*n* [*address*]] | Set index breakpoint |
| C 0\|1 | Set/clear C bit |
| CAPTURE *address* [*address ...*] | Capture changed data |
| CAPTUREFILE [*filename* [R\|A]] | Open capture file |
| CCR *n* | Set condition code register |
| CF [*filename* [R\|A]] | Open capture file |
| CLEARMAP | Clear map file |
| CLEARSYMBOL | Clear user symbols |
| COLORS | Set screen colors |
| COUNTER [*address*] | Set instruction counter |
| CY [*n*] | Set cycles counter |
| CYCLES [*n*] | Set cycles counter |
| DASM [*address* \| *startrange endrange*] | Disassemble memory |
| DDRA *n* | Set port A direction register |
| DDRB *n* | Set port B direction register |
| DUMP[.B\|.W] *startrange endrange* [*n*] | Dump memory to screen |
| EVAL *n* [ *op n*] | Evaluate expression |
| EXIT | Exit program, |
| G [*startaddr* [*endaddr*]] | Begin program execution |
| GO [*startaddr* [*endaddr*]] | Begin program execution |

**Table 3-1. ICS05J1A Commands (continued)**

| Syntax | Meaning |
|---|---|
| GOMACRO *filename* | Execute macro after break |
| GOTIL *endaddr* | Execute until location |
| GOTOCYCLE *n* | Execute until cycle counter value |
| H 0\|1 | Set/clear H bit |
| HELP [*topic*] | Open help system |
| HIDECHIP | Close chip window |
| HISTORY | History capture |
| HISTORYLOG *n* | Log history capture |
| I 0\|1 | Set/clear I bit |
| INFO | Display line information |
| INPUTA *n* | Set port A inputs |
| INPUTB *n* | Set port B inputs |
| INPUTS | Show port inputs |
| INT [0 \| 1] | Set IRQ pin state |
| IRQ [0 \| 1] | Set IRQ pin state |
| LF [*filename* [R\|A]] | Open /close log file |
| LISTOFF | Do not show steps |
| LISTON | Show steps |
| LOAD [*filename*] | Load .S19 file |
| LOADMAP [*filename*] | Load map file |
| LOGFILE[*filename* [R\|A]] | Open /close log file |
| MACRO [*filename*] | Execute script file |
| MACROEND | Stop saving commands |
| MACROSTART *filename* | Save debug commands |
| MD *address* | Display memory |
| MEM *address* [*n...*] | Modify memory |
| MM *address* [*n...*] | Modify memory |
| N 0\|1 | Set/clear N bit |
| NOBR [*address*] | Remove breakpoints |

**Table 3-1. ICS05J1A Commands (continued)**

| Syntax | Meaning |
|---|---|
| NOCNT [*address*] | Remove instruction counters |
| PC *address* | Set PC value |
| POD *n* | Check serial port |
| PORTA *n* | Set port A output latches |
| PORTB *n* | Set port B output latches |
| PROGRAM | Program EPROM |
| PRTA *n* | Set port A output latches |
| PRTB *n* | Set port B output latches |
| QUIET | Suspend window updates |
| QUIT | Exit program |
| R | Set up system registers |
| REG | Show registers |
| REM *text* | Add comment to script file |
| RESET | Simulate processor reset |
| RESET_COUNTS | Reset instruction counters |
| RESETGO | Reset and restart MCU |
| RUN [*startaddr* [*endaddr*]] | Begin program execution |
| SCRIPT [*filename*] | Execute script file |
| SHELL [*command*] | Shell to DOS |
| SHOW *address* | Display memory |
| SHOWCHIP | Open chip window |
| SHOWCODE *address* | Display code |
| SHOWPC | Display code at PC |
| SNAPSHOT | Save screen |
| SOURCE | Toggle code display |
| SP *n* | Set SP value |
| SS [*n*] | Source step |
| ST [*n*] | Single step |
| STACK | Display stack |

**Table 3-1. ICS05J1A Commands (continued)**

| Syntax | Meaning |
|---|---|
| STEP [*n*] | Single step |
| STEPFOR | Step forever |
| STEPTIL *address* | Step until location |
| SYMBOL [*label value*] | Add symbol |
| SYSINFO | Show system information |
| T [*n*] | Single step |
| UPLOAD_SREC *startrange endrange* | Upload S-record |
| VAR[.B\|.W\|.S] *address* [*n*] | Display variable |
| VER | Display version |
| VERSION | Display version |
| WAIT *n* | Wait n cycles |
| WHEREIS *symbol* \| *address* | Display symbol value |
| X *value* | Set X register value |
| XREG *value* | Set X register value |
| Z 0\|1 | Set /clear Z bit |
| ZOOM | Resize source window |

# A Set Accumulator Value A

The **A** command sets the accumulator to a specified value. (The **A** and **ACC** commands are identical.)

**Syntax:**

**A** ***n***

where:

*n* The value to be loaded into the accumulator.

**Example:**

**>A 10** Set the accumulator to 10.

# ACC Set Accumulator Value ACC

The **ACC** command sets the accumulator to a specified value. (The **ACC** and **A** commands are identical.)

**Syntax:**

**ACC *n***

where:

*n* The value to be loaded into the accumulator.

**Example:**

>**ACC 20** Set the accumulator to 20.

# ASM Assemble Instructions ASM

The **ASM** command assembles M68HC05 Family instruction mnemonics, placing the resulting machine code into memory at a specified address.

The command displays the specified address and contents, and prompts for an instruction. As you enter each instruction, the command assembles the instruction, stores and displays the resulting machine code, then moves to the next memory location. To terminate the command, enter a period ( . ).

**Syntax:**

**ASM *address***

where:

| | | |
|---|---|---|
| | *address* | Address for the first machine code generated. |

**Examples:**

```
>ASM 300
 0300        9D    NOP        >CLRA
 0300        4F    CLRA
 0301        9D    NOP        >.
```

# BELL Sound Bell BELL

The **BELL** command sounds the computer bell the specified *hexadecimal* number of times. The bell sounds once if you do not enter an argument. To turn off the bell as it is sounding, press any key.

**Syntax:**

**BELL [*n*]**

where:

*n* The *hexadecimal* number of times to sound the bell.

**Examples:**

| | |
|---|---|
| **>BELL** | Sound the bell once. |
| **>BELL C** | Sound the bell 12 (decimal) times. |
| **>BELL 12** | Sound the bell 18 (decimal) times. |

# BF Block Fill BF

The **BF** command fills a block of memory with a specified byte or word value. The optional variant specifies whether to fill the block in bytes (**.B**, the default) or in words (**.W**).

**Syntax:**

**BF[.B | .W]** ***startrange endrange n***

where:

| | |
|---|---|
| *startrange* | Beginning address of the memory block (range). |
| *endrange* | Ending address of the memory block (range). |
| *n* | Byte or word value to be stored in the specified block. If *n* is an 8-bit value, each byte of the block receives the value. If *n* is a 16-bit value, each word of the block receives the value. |

**Examples:**

| | |
|---|---|
| **>BF C0 CF FF** | Store FF in bytes at addresses C0 — CF. |
| **>BF.W 400 41F 4143** | Store 4143 in words at addresses 400 — 41F. |

# BR Set Instruction Breakpoint BR

The **BR** command displays or sets instruction breakpoints, according to it parameter values:

- With no parameter values, the command displays a list of all current breakpoints in the debug window.
- With an *address* value, the command sets a breakpoint at the specified address.
- With *address* and *n* values, the command sets a breakpoint at the specified address, but code execution does not break until the nth time it arrives at the breakpoint.

**NOTE**

The maximum number of breakpoint addresses is 64. Each **BR**, **BREAKA**, **BREAKSP**, or **BREAKX** command that includes an *address* value uses an additional breakpoint address, unless the address is a duplicate. For example, if 64 **BR** commands already have taken up 64 addresses, the only way to include an *address* value in a **BREAKA**, **BREAKSP**, or **BREAKX** command is to duplicate one of those 64 addresses.

An alternative way to see a list of current breakpoints is to press the **F5** key; this brings up the breakpoint window. (Paragraph 2.2.6 gives more information about this window.)

Note this breakpoint alternative for code showing in the code window: position the cursor on a line of code, then press the **Alt-B** keys. This sets a breakpoint at that line of code (or removes an existing breakpoint from the line.) To remove all breakpoints, use the **NOBR** command.

**Syntax:**

**BR [*address* [*n*]]**

where:

| | |
|---|---|
| *address* | The address for a breakpoint. |
| *n* | Break after value: code execution passes through the breakpoint n-1 times, then breaks the nth time it arrives at the breakpoint. |

**Examples:**

| | |
|---|---|
| **>BR 300** | Set a breakpoint at address 300. |
| **>BR 330 8** | Set a breakpoint at address 330, break on eighth arrival at 330. |

# BREAKA Set Accumulator Breakpoint BREAKA

The **BREAKA** command sets an accumulator breakpoint: it breaks code execution when the value of the accumulator equals the specified *n* value.

- With an *n* value, the command forces a break in execution as soon at the accumulator value equals *n*.
- With *n* and *address* values, the command forces a break in execution when the accumulator value equals *n* and execution arrives at the specified address. (If the accumulator value changes from *n* by the time execution arrives at the address, no break occurs.)

**NOTE**

The maximum number of breakpoint addresses is 64. Each **BR**, **BREAKA**, **BREAKSP**, or **BREAKX** command that includes an *address* value uses an additional breakpoint address, unless the address is a duplicate. For example, if 64 **BR** commands already have taken up 64 addresses, the only way to include an *address* value in a **BREAKA**, **BREAKSP**, or **BREAKX** command is to duplicate one of those 64 addresses.

For a **BREAKA** command without an *address* value, the break in code execution removes the accumulator breakpoint. To cancel the accumulator breakpoint before the break occurs, enter the **BREAKA** command without any parameter values. (If you enter the **BREAKA** command without an *address* value, the accumulator breakpoint does not show in the breakpoint window.)

If you enter the **BREAKA** command *with* an *address* value, you may remove the accumulator breakpoint by (1) entering the **NOBR** command or (2) positioning the cursor on that address in the code window, then pressing **Alt-B**.

**Syntax:**

**BREAKA [*n* [*address*]]**

where:

| | |
|---|---|
| *n* | Accumulator value that triggers a break in execution. |
| *address* | Optional address for the break in execution (provided that the accumulator value equals *n*.) |

**Examples:**

| | |
|---|---|
| >**BREAKA 55** | Break execution when the accumulator value equals 55. |
| >**BREAKA** | Cancel the accumulator breakpoint. |
| >**BREAKA 55 400** | Break execution at address 400 if accumulator value equals 55. |

# BREAKSP Set Stack Pointer Breakpoint BREAKSP

The **BREAKSP** command sets a stack pointer breakpoint: it breaks code execution when the value of the stack pointer equals a specified value.

- With an *n* value, the command forces a break in execution as soon at the stack pointer value equals *n*.
- With *n* and *address* values, the command forces a break in execution when the stack pointer value equals *n* and execution arrives at the specified address. (If the stack pointer value changes from *n* by the time execution arrives at the address, no break occurs.)

**NOTE**

The maximum number of breakpoint addresses is 64. Each **BR**, **BREAKA**, **BREAKSP**, or **BREAKX** command that includes an *address* value uses an additional breakpoint address, unless the address is a duplicate. For example, if 64 **BR** commands already have taken up 64 addresses, the only way to include an *address* value in a **BREAKA**, **BREAKSP**, or **BREAKX** command is to duplicate one of those 64 addresses.

For a **BREAKSP** command without an *address* value, the break in code execution removes the stack pointer breakpoint. To cancel the stack pointer breakpoint before the break occurs, enter the **BREAKSP** command without any parameter values. (If you enter the **BREAKSP** command without an *address* value, the stack pointer breakpoint does not show in the breakpoint window.)

If you enter the **BREAKSP** command *with* an *address* value, you may remove the stack pointer breakpoint by (1) entering the **NOBR** command or (2) positioning the cursor on that address in the code window, then pressing **Alt-B**.

**Syntax:**

**BREAKSP [*n* [*address*]]**

where:

| | |
|---|---|
| *n* | Stack pointer value that triggers a break in execution: $C0 — $FF. |
| *address* | Optional address for the break in execution (provided that the stack pointer value equals *n*.) |

**Examples:**

| | |
|---|---|
| **>BREAKSP E0** | Break execution when the SP value equals E0. |
| **>BREAKSP** | Cancel SP breakpoint. |
| **>BREAKA E0 400** | Break execution at address 400 if SP value equals E0. |

# BREAKX Set Index Breakpoint BREAKX

The **BREAKX** command sets an index breakpoint: it breaks code execution when the value of the index register equals a specified value.

- With an *n* value, the command forces a break in execution as soon at the index register value equals *n*.
- With *n* and *address* values, the command forces a break in execution when the index register value equals *n* and execution arrives at the specified address. (If the index register value changes from *n* by the time execution arrives at the address, no break occurs.)

**NOTE**

The maximum number of breakpoint addresses is 64. Each **BR**, **BREAKA**, **BREAKSP**, or **BREAKX** command that includes an *address* value uses an additional breakpoint address, unless the address is a duplicate. For example, if 64 **BR** commands already have taken up 64 addresses, the only way to include an *address* value in a **BREAKA**, **BREAKSP**, or **BREAKX** command is to duplicate one of those 64 addresses.

For a **BREAKX** command without an *address* value, the break in code execution removes the index breakpoint. To cancel the index breakpoint before the break occurs, enter the **BREAKX** command without any parameter values. (If you enter the **BREAKX** command without an *address* value, the index breakpoint does not show in the breakpoint window.)

If you enter the **BREAKX** command *with* an *address* value, you may remove the index breakpoint by (1) entering the **NOBR** command or (2) positioning the cursor on that address in the code window, then pressing **Alt-B**.

**Syntax:**

**BREAKX [*n* [*address*]]**

where:

| | |
|---|---|
| *n* | Index register value that triggers a break in execution. |
| *address* | Optional address for the break in execution (provided that the index register value equals *n*.) |

**Examples:**

| | |
|---|---|
| **>BREAKX A9** | Break execution when the index register value equals A9. |
| **>BREAKX** | Cancel index breakpoint. |
| **>BREAKX A9 400** | Break execution at address 400 if index register value equals A9. |

# C Set/Clear C Bit C

The **C** command sets or clears (that is, assigns *0* or *1* to) the C bit of the condition code register (CCR).

**NOTE**

The CCR bit designators are at the lower right of the CPU window. The CCR pattern is 111HINZC (H is half-carry, I is IRQ interrupt mask, N is negative, Z is zero, and C is carry). A letter in these designators means that the corresponding bit of the CCR is set; a period means that the corresponding bit is clear.

**Syntax:**

```
C 0|1
```

**Examples:**

| | |
|---|---|
| **>C 0** | Clear the C bit of the CCR. |
| **>C 1** | Set the C bit of the CCR. |

# CAPTURE Capture Changed Data CAPTURE

The **CAPTURE** command specifies locations to be monitored for changes in value. If the value of such a location changes and if a capture file is open, the file records the change in value. (Paragraph 2.8 gives more information about capture files.)

To stop monitoring a location, specify that same location in another **CAPTURE** command, or close the capture file. (Closing the capture file undoes the specifications for all monitoring locations.)

**NOTE**

Before you enter the **CAPTURE** command, open a capture file via the **CAPTUREFILE** or **CF** command. The **CAPTURE** command has no effect unless a capture file is open.

**Syntax:**

**CAPTURE *address* [*address ...*]**

where:

*address* Location to be monitored for a change in value.

**Examples:**

>**CAPTURE PORTA** Monitor location PORTA for any value changes.

>**CAPTURE C0** Monitor RAM location C0 for any value changes.

>**CAPTURE D0 D1 D2** Monitor for any value changes in an array of locations.

# CAPTUREFILE Open Capture File CAPTUREFILE

The **CAPTUREFILE** command opens a capture file to record changed values. If the specified file does not yet exist, this command creates the file. (Paragraph 2.8 gives more information about capture files.)

If the file already exists, you can use an optional parameter to specify whether to overwrite existing contents (**R**, the default) or to append the log entries (**A**). If you omit this parameter, a prompt asks for this overwrite/append choice.

To close the capture file, enter this command without any parameter values.

**NOTES**

The **CAPTURE** command specifies the locations to be monitored for value changes. Closing the capture file deletes the location specification.

The simulator continues writing to an open capture file. You must close the capture file within a reasonable time, to prevent the file from growing large.

(The **CF** and **CAPTUREFILE** commands are identical. If no **CAPTURE** command has specified locations to be monitored, the **CF** and **CAPTUREFILE** commands have no effect.)

**Syntax:**

**CAPTUREFILE [*filename* [R | A]]**

where:

*filename* Name of the capture file.

**Examples:**

| | |
|---|---|
| **>CAPTUREFILE TEST.CAP** | Open capture file TEST.CAP. |
| **>CAPTUREFILE TEST4.CAP A** | Open capture file TEST4.CAP; append new entries. |

# CCR Set Condition Code Register CCR

The CCR command sets the condition code register (CCR) to the specified *hexadecimal* value.

**NOTE**

The CCR bit designators are at the lower right of the CPU window. The CCR binary pattern is 111HINZC (H is half-carry, I is IRQ interrupt mask, N is negative, Z is zero, and C is carry). A letter in these designators means that the corresponding bit of the CCR is set; a period means that the corresponding bit is clear.

**Syntax:**

**CCR** ***n***

where:

*n* The new *hexadecimal* value for the CCR.

**Example:**

**>CCR E4** Assign the value *E4* to the CCR. (This makes the binary pattern *11100100*: the N bit set, other bits clear).

# CF Open Capture File CF

The **CF** command opens a capture file to record changed values. If the specified file does not yet exist, this command creates the file. (Paragraph 2.8 gives more information about capture files.)

If the file already exists, you can use an optional parameter to specify whether to overwrite existing contents (**R**, the default) or to append the log entries (**A**). If you omit this parameter, a prompt asks for this overwrite/append choice.

To close the capture file, enter this command without any parameter values.

**NOTES**

The **CAPTURE** command specifies the locations to be monitored for value changes. Closing the capture file deletes the location specification.

The simulator continues writing to an open capture file. You must close the capture file within a reasonable time, to prevent the file from growing large.

(The **CF** and **CAPTUREFILE** commands are identical. If no **CAPTURE** command has specified locations to be monitored, the **CF** and **CAPTUREFILE** commands have no effect.)

**Syntax:**

**CF [*filename* [R | A]]**

where:

*filename* Name of the capture file.

**Examples:**

>**CF TEST.CAP** Open capture file `TEST.CAP`.

>**CF TEST4.CAP A** Open capture file `TEST4.CAP`; append new entries.

# CLEARMAP Clear Map File CLEARMAP

The **CLEARMAP** command removes the current loaded source-level map file from the system.

(Use the **LOAD** or **LOADMAP** command to load a map file.)

**Syntax:**

**CLEARMAP**

**Example:**

>**CLEARMAP** Remove current source-level map file.

# CLEARSYMBOL Clear User Symbols CLEARSYMBOL

The **CLEARSYMBOL** command removes all user-defined symbols. This command does not affect **MAP** file debug information.

(Use the **SYMBOL** command to define symbols. For a list of current user-defined symbols, enter the **SYMBOL** command without any parameter values.)

**Syntax:**

**CLEARSYMBOL**

**Example:**

>**CLEARSYMBOL** Clear user-defined symbols.

# COLORS Set Screen Colors COLORS

The **COLORS** command sets the screen colors. Entering this command brings up the color selection window: a matrix of foreground/background color combinations and a list of screen elements. Each color combination has a two-digit hexadecimal number; the first screen element is highlighted.

Work your way down the list of screen elements by entering the hexadecimal number of the color combination you want for the element, then pressing **Enter**. (To retain the current color combination, press **Enter** without entering a new number.) As the software accepts your selection for the current screen element, it highlights the next element. The **COLORS** command ends when you select a color combination for the last screen element, or when you press **ESC**.

In the color matrix, rows correspond to the eight background colors; columns correspond to the 16 foreground colors.

**Syntax:**

**COLORS**

**Example:**

>**COLORS** Set screen colors.

# COUNTER Set Instruction Counter COUNTER

The **COUNTER** command sets (or removes) an instruction counter at the specified address. If no counter is set at the address already, this command sets one. If a counter already is set at the address, this command removes the counter. The maximum number of instruction counters is 32; you may not set an instruction counter at a breakpoint address.

When you set a new instruction counter, it has the initial value *0*. Each time code execution passes through the counter, the software increments the counter value. To clear (reassign the value *0* to) all the instruction counters, enter the **RESET_COUNTS** command. To remove all the instruction counters, enter the **NOCNT** command.

To see a list of current instruction counters, in the debug window, enter the **COUNTER** command without an *address* value. An alternative way to see a list of current instruction counters is to press the **F6** key; this brings up the instruction counter window. (Paragraph 2.2.6 gives more information about this window.)

Note this instruction-counter alternative for code showing in the code window: position the cursor on a line of code, then press the **Alt-C** keys. This sets a counter at that line of code (or removes an existing counter from the line.)

**Syntax:**

**COUNTER [*address*]**

where:

*address* The address for an instruction counter.

**Examples:**

| | |
|---|---|
| >**COUNTER 300** | Set an instruction counter at location 300. |
| >**COUNTER** | Show current instruction counters in debug window. |

# CY Set Cycles Counter CY

The **CY** command manages the cycles counter. The cycles counter counts the number of MCU cycles; the count can be useful for timing or scheduling.

- If you enter the **CY** command without any parameter value, it displays the current cycle count in hexadecimal and decimal formats, and relates the count to a 2-megahertz bus speed.
- If you enter the **CY** command with an *n* parameter value, it assigns that value to the cycles counter

(The **CYCLES** command is identical.)

**Syntax:**

**CY [*n*]**

where:

| | |
|---|---|
| *n* | Integer value for the cycles counter. |

**Examples:**

| | |
|---|---|
| >**CY 0** | Reset cycles counter. |
| >**CY** | Show cycle count, relate to 2-megahertz bus speed. |
| >**CY 1000** | Set cycle-counter value to 1000. |

# CYCLES Set Cycles Counter CYCLES

The **CYCLES** command manages the cycles counter. The cycles counter counts the number of MCU cycles; the count can be useful for timing or scheduling.

- If you enter the **CYCLES** command without any parameter value, it displays the current cycle count in hexadecimal and decimal formats, and relates the count to a 2-megahertz bus speed.
- If you enter the **CYCLES** command with an *n* parameter value, it assigns that value to the cycles counter

(The **CY** command is identical.)

**Syntax:**

**CYCLES [*n*]**

where:

| | |
|---|---|
| *n* | Integer value for the cycles counter. |

**Examples:**

| | |
|---|---|
| >**CYCLES 0** | Reset cycles counter. |
| >**CYCLES** | Show cycle count, relate to 2-megahertz bus speed. |
| >**CYCLES 1000** | Set cycle-counter value to 1000. |

# DASM Disassemble Memory DASM

The **DASM** command disassembles machine instructions, displaying the addresses and the contents as disassembled instructions in the debug window:

- If the command includes an *address* value, ICS05J1A shows three disassembled instructions, beginning at that address.
- If you enter the **DASM** command without any parameter values, the software finds the most recently disassembled instruction, then shows the next three instructions, disassembled.
- If the command includes *startrange* and *endrange* values, the software shows disassembled instructions for the range.

**NOTE**

If you enter the **DASM** command for a range, the disassembled instructions scroll through the debug window too rapidly to view. Accordingly, you first should enter the **LOGFILE** command, which records the disassembled instructions in a log.

**Syntax:**

**DASM [*address* | *startrange endrange*]**

where:

| | |
|---|---|
| *address* | First address of three instruction opcodes to be disassembled. |
| *startrange* | Starting address for a range of instructions to be disassembled. |
| *endrange* | Ending address for a range of instructions to be disassembled. |

**Examples:**

```
>DASM 300
0300          A6E8        LDA #0E8
0302          B702        STA 0002
0304          4F          CLRA

>DASM 400 408
0400          5F          CLRX
0401          A680        LDA #80
0403          B700        STA PORTA
0405          A6FE        LDA #FE
0407          B704        STA DDRA
```

# DDRA Set Port A Direction Register DDRA

The **DDRA** command assigns the specified byte value to the port A data direction register (DDR).

**Syntax:**

**DDRA** ***n***

where:

*n* Byte value for the port A DDR.

**Examples:**

**>DDRA FF** Set all port A pins to be outputs.

**>DDRA 00** Set all port A pins to be inputs.

# DDRB Set Port B Direction Register DDRB

The **DDRB** command assigns the specified byte value to the port B data direction register (DDR).

**NOTE**

The DDRB is a 6-bit register. The **DDRB** command ignores the upper two bits of larger values.

**Syntax:**

**DDRB** ***n***

where:

*n* Byte value for the port B DDR.

**Examples:**

**>DDRB 3F** Set all port B pins to be outputs.

**>DDRB FF** Set all port B pins to be outputs (ignoring the upper two bits).

# DUMP Dump Memory to Screen DUMP

The **DUMP** command sends contents of a block of memory to the debug window, in bytes or words. The optional variant specifies whether to fill the block in bytes (**.B**, the default) or in words (**.W**).

**NOTE**

If you enter the **DUMP** command, the memory contents scroll through the debug window too rapidly to view. Accordingly, you first should enter the **LOGFILE** command, which records the memory contents in a log.

**Syntax:**

**DUMP[.B | .W]** ***startrange endrange*** **[*n*]**

where:

| | |
|---|---|
| *startrange* | Beginning address of the memory block (range). |
| *endrange* | Ending address of the memory block (range). |
| *n* | Optional number of bytes or words to be written on one line. |

**Examples:**

| | |
|---|---|
| **>DUMP C0 CF** | Dump array of RAM values, in bytes. |
| **>DUMP.W 400 47F** | Dump ROM code in addresses 400 — 47F, in words. |
| **>DUMP.B 300 400 8** | Dump contents of addresses 300 — 400, in rows of eight bytes. |

# EVAL Evaluate Expression EVAL

The **EVAL** command evaluates a numerical term or simple expression, giving the result in hexadecimal, decimal, octal, binary, and ASCII-character formats. In an expression, spaces must separate the operator from the numerical terms.

(Note that octal numbers are not valid as operand values. Operand values are 16 bits or less.) If the value is printable, this command also displays the value in ASCII characters. The operand can be a number or the sequence number, space, operator, space, and number. This command supports addition (+), subtraction (-), multiplication (*) and division (/).

**Syntax:**

**EVAL *n*[ *op n*]**

where:

| | |
|---|---|
| *n* | Alone, the numerical term to be evaluated. Otherwise, either numerical term of a simple expression. |
| *op* | The arithmetic operator (+, -, *, or /) of a simple expression. |

**Examples:**

```
>EVAL 45 + 32
0077H 119T 000167O 0000000001110111Q "w"
>EVAL 100T
0064H 100T 000144O 0000000001100100Q "d"
```

# EXIT Exit Program EXIT

The **EXIT** command terminates the host session and returns to DOS or RAPID. (The **QUIT** command is identical.)

**Syntax:**

**EXIT**

**Example:**

>**EXIT** Return to DOS or RAPID.

# G Begin Program Execution G

The **G** command starts execution of code in the simulator, beginning at the current program counter (PC) address, or at an optional *startaddr* address. Execution continues until it encounters a breakpoint, until it encounters an error, or until you press a key.

If the **G** command includes both *startaddr* and *endaddr* values, code execution begins at the *startaddr* address and ends at the *endaddr* address.

The **GO** and **RUN** commands are optional forms of the **G** command.

**Syntax:**

**G [*startaddr* [*endaddr*]]**

where:

| | |
|---|---|
| *startaddr* | Optional execution starting address. If the command does not have a startaddr value, execution begins at the current PC value. |
| *endaddr* | Optional execution ending address. |

**Examples:**

| | |
|---|---|
| **>G** | Begin code execution at the current PC value. |
| **>G 346** | Begin code execution at address 346. |
| **>G 400 471** | Begin code execution at address 400. End code execution just before the instruction at address 471. |

# GO Begin Program Execution GO

The **GO** command, like the **RUN** command, is an optional form of the **G** command.

**Syntax:**

**GO [*startaddr* [*endaddr*]]**

where:

| | |
|---|---|
| *startaddr* | Optional execution starting address. If the command does not have a startaddr value, execution begins at the current PC value. |
| *endaddr* | Optional execution ending address. |

# GOMACRO Execute Macro after Break GOMACRO

The **GOMACRO** command starts execution of code in the simulator, beginning at the current program counter (PC) address. Execution continues until it encounters a breakpoint, when the software begins execution of the specified macro.

**Syntax:**

**GOMACRO *filename***

where:

| | |
|---|---|
| *filename* | Name of the script file to be executed as a macro. The *filename* value need not include the .MAC extension. The *filename* value can be a pathname that includes an asterisk (*) wildcard character; if so, the software displays a list of possible macro files, for your selection. |

**Example:**

| | |
|---|---|
| **>GOMACRO AVCALC.MAC** | Begin code execution at the current PC value; at breakpoint execute macro AVCALC.MAC. |

# GOTIL Execute until Location GOTIL

The **GOTIL** command executes the program in the emulator, beginning at the address in the program counter (PC). Execution continues until the instruction pointer contains the specified ending address.

Pressing the **Alt-G** keys is similar: code executes from the address in the PC to the highlighted line in the code window.

**Syntax:**

**GOTIL *endaddr***

where:

*endaddr* Execution stop address.

**Example:**

**>GOTIL 3F0** Execute the program in the emulator up to address 3F0.

# GOTOCYCLE Execute until Cycle Counter Value GOTOCYCLE

The **GOTOCYCLE** command executes the program in the emulator, beginning at the address in the program counter (PC). Execution continues until the simulator's cycle counter contains the specified value.

**Syntax:**

**GOTOCYCLE *n***

where:

*n* Cycle-counter value at which execution stops.

**Example:**

>**GOTOCYCLE 100** Execute the program until the cycle count equals 100.

# H Set/Clear H Bit H

The **H** command sets or clears (that is, assigns *0* or *1* to) the H bit of the condition code register (CCR).

**NOTE**

The CCR bit designators are at the lower right of the CPU window. The CCR pattern is 111HINZC (H is half-carry, I is IRQ interrupt mask, N is negative, Z is zero, and C is carry). A letter in these designators means that the corresponding bit of the CCR is set; a period means that the corresponding bit is clear.

**Syntax:**

```
H 0|1
```

**Examples:**

| | |
|---|---|
| **>H 0** | Clear the H bit of the CCR. |
| **>H 1** | Set the H bit of the CCR. |

# HELP Open Help System HELP

The **HELP** command opens the ICS05J1A pop-up help system. If you enter this command with an optional command name, help information for that command appears on the screen. If you enter this command without any parameter value, a help menu appears; you may select a topic from this menu.

An alternative way to open the help system is to press the **F1** key. If you type an ICS05J1A command in the debug window, but press **F1** instead of **Enter**, you bring up help information about the command immediately, by-passing the help menu.

To return to the help menu, press **Esc**; to exit the help system, press **Esc** again.

**Syntax:**

**HELP [*topic*]**

where:

*topic* An ICS05J1A software command or HC05 instruction.

**Examples:**

| | |
|---|---|
| **>HELP** | Open the help system. |
| **>HELP GO** | Open **GO**-command help information. |

# HIDECHIP Close Chip Window HIDECHIP

The **HIDECHIP** command closes the chip visualization window, restoring the variables window.

(The **SHOWCHIP** command opens the chip visualization window.)

**Syntax:**

**HIDECHIP**

**Example:**

**>HIDECHIP** Close chip visualization window.

# HISTORY History Capture HISTORY

The **HISTORY** command enables or disables history capture. When you enable history capture, the simulator records instructions, timer values, and CPU values in a 1024-element circular buffer.

While history capture is enabled, the word `HISTORY` appears in the center of the main screen. Note that history capture somewhat slows execution.

The simulator disassembles captured history information when you view buffer contents. To view buffer contents, press **F4**; contents appear in the history window. (If you never have enabled history capture, the history window shows the indication `NO HISTORY`.) To clear the history buffer, toggle it OFF and ON.

To write history-buffer contents to a log file, use the **HISTORYLOG** command in conjunction with the **HISTORY** and **LOGFILE** (or **LF**) commands.

**Syntax:**

**HISTORY**

**Example:**

>**HISTORY** Enable (or disable) history capture.

# HISTORYLOG Log History Capture HISTORYLOG

The **HISTORYLOG** command adds history-buffer contents to a log file, provided that a log file is open, and provided that history capture is enabled. The *n* parameter specifies the number of most recently captured items to log.

**NOTE**

To open a log file, use the **LOGFILE** or **LF** command. To enable history capture, use the **HISTORY** command. The **HISTORYLOG** command should be the third command of this sequence.

**Syntax:**

**HISTORYLOG *n***

where:

*n* Number of history-buffer instructions to log.

**Example:**

>**HISTORYLOG 10** Log most recent 16 (decimal) instructions from the history buffer.

## I — Set/Clear I Bit

The **I** command sets or clears (that is, assigns *0* or *1* to) the I bit of the condition code register (CCR).

**NOTE**

The CCR bit designators are at the lower right of the CPU window. The CCR pattern is 111HINZC (H is half-carry, I is IRQ interrupt mask, N is negative, Z is zero, and C is carry). A letter in these designators means that the corresponding bit of the CCR is set; a period means that the corresponding bit is clear.

**Syntax:**

```
I 0|1
```

**Examples:**

| | |
|---|---|
| **>I 0** | Clear the I bit of the CCR. |
| **>I 1** | Set the I bit of the CCR. |

# INFO Display Line Information INFO

The **INFO** command displays information about the cursor line in the source window. This information includes the name of the file being displayed in the window, the line number, the address, the corresponding object code, and the disassembled instruction.

**Syntax:**

**INFO**

**Example:**

>**INFO** Display information about the cursor line.

```
Filename     :    PODTEST.ASM     Line number     :    6
Address      :    $0300
Disassembly  :    START           5F                   CLRX
```

# INPUTA Set Port A Inputs INPUTA

The **INPUTA** command sets the simulated inputs to port A. The CPU reads this simulated input for LDA PRTA or a similar instruction.

**NOTE**

If the JICS circuit board is connected , port A inputs come from the board, so this command has no effect.

**Syntax:**

**INPUTA *n***

where:

*n* Eight-bit simulated value for port A.

**Example:**

**>INPUTA AA** Simulate the input *AA* on port A.

# INPUTB Set Port B Inputs INPUTB

The **INPUTB** command sets the simulated inputs to port B. The CPU reads this simulated input for LDA PRTB or a similar instruction.

**NOTES**

If the JICS circuit board is connected , port B inputs come from the board, so this command has no effect.

Port B consists of six bits; the **INPUTB** command ignores the upper two bits of larger values.

**Syntax:**

**INPUTB *n***

where:

*n* Six-bit simulated value for port B.

**Example:**

**>INPUTB 11** Simulate the input *11*on port B.

# INPUTS Show Port Inputs INPUTS

The **INPUTS** command shows the simulated input values to ports A and B (entered via the **INPUTA** and **INPUTB** commands).

**NOTE**

If the JICS circuit board is connected , this command shows values from the board.

**Syntax:**

**INPUTS**

**Example:**

>**INPUTS** Show simulated input values.

```
Port A - AA  Port B - 01
```

# INT Set IRQ Pin State INT

The **INT** command assigns the state value of the MCU IRQ pin. To see the current simulated value on the pin, enter this command without any parameter value. (The **IRQ** command is identical.)

**NOTE**

If the JICS circuit board is connected , the IRQ pin value comes from the board, so this command has no effect.

**Syntax:**

**INT [0|1]**

**Examples:**

| | |
|---|---|
| **>INT 0** | Assign 0 to the IRQ pin. |
| **>INT 1** | Assign 1 to the IRQ pin. |

# IRQ — Set IRQ Pin State — IRQ

The **IRQ** command assigns the state value of the MCU IRQ pin. To see the current simulated value on the pin, enter this command without any parameter value. (The **INT** command is identical.)

**NOTE**

If the JICS circuit board is connected , the IRQ pin value comes from the board, so this command has no effect.

**Syntax:**

```
IRQ [0|1]
```

**Examples:**

| | |
|---|---|
| **>IRQ 0** | Assign 0 to the IRQ pin. |
| **>IRQ 1** | Assign 1 to the IRQ pin. |

# LF Open/Close Log File LF

The **LF** command opens an external file to receive log entries of commands and copies of responses in the debug window. If the specified file does not yet exist, this command creates the file. (The **LOGFILE** command is identical.)

If the file already exists, you can use an optional parameter to specify whether to overwrite existing contents (**R**, the default) or to append the log entries (**A**). If you omit this parameter, a prompt asks for this overwrite/append choice.

While logging remains in effect, any line that is appended to the debug window is also written to the log file. Logging continues until you enter another **LF** or **LOGFILE** command *without any parameter values*; this second command disables logging and closes the log file.

**Syntax:**

**LF [*filename* [R|A]]**

where:

| | |
|---|---|
| *filename* | The DOS filename of the log file (or logging device). |

**Examples:**

| | |
|---|---|
| >**LF TEST.LOG** | Start logging. Overwrite file `TEST.LOG` (in the current directory) with all lines that appear in the debug window. |
| >**LF TEMP.LOG A** | Start logging. Append to file `TEST.LOG` (in the current directory) all lines that appear in the debug window. |
| >**LF** | (If logging is enabled): Disable logging and close the log file. |

# LISTOFF Do Not Show Steps LISTOFF

The **LISTOFF** command turns off a screen listing of step-by-step information for stepping. Register values and program instructions do not appear in the debug window as code runs. (This display state is the default when you run ICS05J1A software.)

To turn on the display of stepping information, use the **LISTON** command.

**Syntax:**

**LISTOFF**

**Example:**

>**LISTOFF** Do not show step information.

# LISTON Show Steps LISTON

The **LISTON** command shows a screen listing of step-by-step information during stepping. Register values and program instructions appear in the debug window as code runs.

To turn off this step display, use the **LISTOFF** command.

**Syntax:**

**LISTON**

**Example:**

>**LISTON** Show step information.

# LOAD Load S19 File LOAD

The **LOAD** command loads the specified .S19-format file (and any map file with the same name) into the simulator.

Entering this command without a *filename* value brings up a list of .S19 files in the current directory. You can select a file to be loaded directly from this list.

**Syntax:**

**LOAD [*filename*]**

where:

*filename* The name of the .S19 file to be loaded. The *filename* value need not include the .S19 extension. The *filename* value can be a pathname that includes an asterisk (*) wildcard character; if so, the software displays a list of possible files, for your selection.

**Examples:**

**>LOAD PROG1.S19** Load file PROG1.S19 and its map file into the emulator at the load addresses in the file.

**>LOAD PROG2** Load file PROG2.S19 and its map file into the emulator at the load addresses in the file.

**>LOAD A:*** Display the names of the .S19 files on the diskette in drive A:, for user selection of a file.

**>LOAD** Display the names of the .S19 files in the current directory, for user selection of a file.

# LOADMAP Load Map File LOADMAP

The **LOADMAP** command loads a map file that contains source level debug information into the host computer.

Entering this command without a *filename* value brings up a list of .MAP files in the current directory. You can select a file to be loaded directly from this list.

**Syntax:**

**LOADMAP [*filename*]**

where:

| | |
|---|---|
| *filename* | The name of the map file to be loaded. The *filename* value need not include the .MAP extension. The *filename* value can be a pathname that includes an asterisk (*) wildcard character; if so, the software displays a list of possible .MAP files, for your selection. |

**Examples:**

| | |
|---|---|
| **>LOADMAP PROG1.MAP** | Load map file PROG1.MAP into the host computer. |
| **>LOADMAP PROG2** | Load map file PROG2.MAP into the host computer. |
| **>LOADMAP A:*** | Display the names of the .MAP files on the diskette in drive A:, for user selection of a file. |
| **>LOADMAP** | Display the names of the .MAP files in the current directory, for user selection of a file. |

# LOGFILE Open/Close Log File LOGFILE

The **LOGFILE** command opens an external file to receive log entries of commands and copies of responses in the debug window. If the specified file does not yet exist, this command creates the file. (The **LF** command is identical.)

If the file already exists, you can use an optional parameter to specify whether to overwrite existing contents (**R**, the default) or to append the log entries (**A**). If you omit this parameter, a prompt asks for this overwrite/append choice.

While logging remains in effect, any line that is appended to the debug window is also written to the log file. Logging continues until you enter another **LF** or **LOGFILE** command *without any parameter values*; this second command disables logging and closes the log file.

**Syntax:**

**LOGFILE [*filename* [R|A]]**

where:

| | |
|---|---|
| *filename* | The DOS filename of the log file (or logging device). |

**Examples:**

| | |
|---|---|
| **>LOGFILE TEST.LOG** | Start logging. Overwrite file TEST.LOG (in the current directory) with all lines that appear in the debug window. |
| **>LOGFILE TEMP.LOG A** | Start logging. Append to file TEST.LOG (in the current directory) all lines that appear in the debug window. |
| **>LOGFILE** | (If logging is enabled): Disable logging and close the log file. |

# MACRO Execute Script File MACRO

The **MACRO** command executes a script file: a file that contains a sequence of debug commands. Executing the script file has the same effect as executing the individual commands, one after another. (The **SCRIPT** command is identical.)

Entering this command without a *filename* value brings up a list of script (.MAC) files in the current directory. You can select a file to be executed directly from this list.

Note that a script file can contain the **SCRIPT** or **MACRO** command; in this way, you can nest script files as many as 16 levels deep.

Also note that the most common use of the **REM** and **WAIT** commands is within script files. The **REM** command displays your comment while the script file executes; the **WAIT** command establishes a pause between the execution of the script-file commands.

If you give a script file the filename `STARTUP.05J`, startup routines run the script file each time you start the ICS05J1A.

**Syntax:**

**MACRO [*filename*]**

where:

| | |
|---|---|
| *filename* | The name of a script file. The *filename* value need not include the .MAC extension. The *filename* value can be a pathname that includes an asterisk (*) wildcard character; if so, the software displays a list of script files, for your selection. |

**Examples:**

| | |
|---|---|
| **>MACRO INIT.MAC** | Execute commands in file INIT.MAC. |
| **>MACRO *** | Display names of all .MAC files (then execute the selected file). |
| **>MACRO A:*** | Display names of all .MAC files in drive A (then execute the selected file). |
| **>MACRO** | Display names of all .MAC files in the current directory, then execute the selected file). |

# MACROEND Stop Saving Commands MACROEND

The **MACROEND** command closes the script file in which the software has saved debug commands. (The **MACROSTART** command opened the script file.]

**Syntax:**

**MACROEND**

**Example:**

**>MACROEND** Stop saving debug commands to the script file; close the file.

# MACROSTART Save Debug Commands MACROSTART

The **MACROSTART** command opens a script file and begins saving subsequent debug commands to that file, for later use. (Before you end the ICS05J1A session, you must use the **MACROEND** command to close the script file.)

**Syntax:**

**MACROSTART *filename***

where:

*filename* The name of the script file to save commands. The *filename* value need not include the .MAC extension. The *filename* value can be a pathname that includes an asterisk (*) wildcard character; if so, the software displays a list of .MAC files, for your selection.

**Example:**

**>MACROSTART TEST.MAC** Save debug commands in script file TEST.MAC.

# MD Display Memory MD

The **MD** command displays (in the memory window) the contents of 32 controller memory locations, beginning at the specified address. If a log file is open, this command also writes the first 16 values to the log file. (The **SHOW** command is identical.)

**Syntax:**

**MD *address***

where:

*address* The starting memory address for display in the memory window.

**Example:**

**>MD 300** Display the contents of 32 bytes of memory beginning at address 300.

# MEM Modify Memory MEM

The **MEM** command modifies contents of memory locations, beginning at the specified *address* value. (The **MM** command is an alternate form of this command.)

If the **MEM** command includes optional data values, the software assigns those values to successive addresses, beginning at the specified address. Then the command ends.

If you do not include any data values, the **MEM** command assigns values interactively. A prompt shows the current value at the specified address and asks for a data value. Enter the data value and press **Enter**; the software assigns the value to the location, then moves to the next address. To leave a memory value unchanged, press **Enter** without entering a new value. To stop assigning memory values, enter a period ( **.** ) instead of a valid value.

**Syntax:**

**MEM *address* [*n...* ]**

where:

| | |
|---|---|
| *address* | The address of the first memory location to be modified. |
| *n* | Optional data value. |

**Examples:**

| | |
|---|---|
| **>MEM D0 11 22 33** | Assign values 11, 22, and 33 to successive addresses D0, D1, and D2. |
| **>MEM 300** | Start memory modify at address 300. |
| 300 = 0F >**05** | Change current 0F value to 05, advance to address 301. |
| 301 = 10 > | Retain current 10 value, advance to address 302. |
| 302 = 22 >**1B** | Change current 22 value to 1B, advance to address 303. |
| 303 = 3D >**.** | End memory modify. |

# MM Modify Memory MM

The **MM** command is an alternative form of the **MEM** command.

**Syntax:**

**MM *address* [*n*... ]**

where:

| | |
|---|---|
| *address* | The address of the first memory location to be modified. |
| *n* | Optional data value. |

# N Set/Clear N Bit N

The **N** command sets or clears (that is, assigns *0* or *1* to) the N bit of the condition code register (CCR).

**NOTE**

The CCR bit designators are at the lower right of the CPU window. The CCR pattern is 111HINZC (H is half-carry, I is IRQ interrupt mask, N is negative, Z is zero, and C is carry). A letter in these designators means that the corresponding bit of the CCR is set; a period means that the corresponding bit is clear.

**Syntax:**

```
N 0|1
```

**Examples:**

| | |
|---|---|
| **>N 0** | Clear the N bit of the CCR. |
| **>N 1** | Set the N bit of the CCR. |

# NOBR Remove Breakpoints NOBR

The **NOBR** command removes one or all addressed breakpoints. If this command has an *address* value, it removes the breakpoint at that address. If this command has no parameter values, it removes all current addressed breakpoints.

Note this alternative for code showing in the code window: position the cursor on a line of code that has a breakpoint, then press the **Alt-B** keys. This removes the breakpoint from the line.

**NOTE**

An *addressed* breakpoint is one set by a **BR**, **BREAKA**, **BREAKSP**, or **BREAKX** command that includes an *address* value. Neither the **NOBR** command nor the **Alt-B** alternative applies to breakpoints set by **BREAKA**, **BREAKSP**, or **BREAKX** commands that do not include *address* values.

**Syntax:**

**NOBR [*address*]**

where:

*address* Optional address of a single breakpoint to be removed.

**Examples:**

| | |
|---|---|
| **>NOBR** | Remove all current addressed breakpoints. |
| **>NOBR 320** | Remove the breakpoint at address 320. |

# NOCNT Remove Instruction Counters NOCNT

The **NOCNT** command removes one or all instruction counters. If this command has an *address* value, it removes the counter at that address. If this command has no parameter values, it removes all current counters.

Note this alternative for code showing in the code window: position the cursor on a line of code that has an instruction counter, then press the **Alt-C** keys. This clears the counter from the line.

To set (create) instruction counters, use the **COUNTER** command. To reset (that is, assign the value 0) to instruction counters, use the **RESET_COUNTS** command.

**Syntax:**

**NOCNT [*address*]**

where:

*address* Optional address of a single instruction counter to be removed.

**Examples:**

| | |
|---|---|
| **>NOCNT** | Remove all current instruction counters. |
| **>NOCNT 420** | Remove the instruction counter at address 420. |

# PC Set PC Value PC

The **PC** command assigns the specified value to the program counter (PC). As the PC always points to the next instruction to be executed, assigning a new PC value changes the flow of code execution.

Note this alternative for code showing in the code window: position the cursor on a line of code, then press the **Alt-P** keys. This assigns the address of that line to the PC.

**Syntax:**

**PC *address***

where:

*address* The new PC value.

**Example:**

**>PC 0500** Set the PC value to *0500*.

# POD Check Serial Port POD

The **POD** command checks the specified serial port for the JICS circuit board. The command responds with the current status of port A, port B, reset, and IRQ on the board. The command also shows the version of the board.

**Syntax:**

**POD n**

where:

*n* The number, 1 — 8, of a serial port.

**Example:**

>**POD 1** Check serial port COM1.

```
Port A - 80  Port B - 00  Reset - 1  Irq - 1  Version - 1.00
```

# PORTA Set Port A Output Latches PORTA

The **PORTA** command assigns the specified value to the port A output register latches. (The **PRTA** command is an alternate form of the **PORTA** command.)

**NOTE**

If the JICS circuit board is connected , the system sends the *n* parameter value of this command to the board.

**Syntax:**

**PORTA** ***n***

where:

*n* The new value for the port A output register latches.

**Example:**

**>PORTA FF** Set all port A output latches high.

# PORTB Set Port B Output Latches PORTB

The **PORTB** command assigns the specified value to the port B output register latches. (The **PRTB** command is an alternate form of the **PORTB** command.)

**NOTES**

If the JICS circuit board is connected , the system sends the *n* parameter value of this command to the board.

Port B consists of six bits; the **PORTB** command ignores the upper two bits of larger values.

**Syntax:**

**PORTB** ***n***

where:

*n* The new value for the port B output register latches.

**Example:**

**>PORTB 33** Set the port B output latches to 33.

# PROGRAM Program EPROM PROGRAM

The **PROGRAM** command starts the EPROM programmer software (which also can program the MOR byte).

Entering this command brings up the programming screen, which consists of three windows:

- **Log window —** which contains instructions for programming. Follow these instructions for installing an MCU into a programming socket, selecting programming actions from the pick window, and removing the programmed MCU.
- **Pick window —** which lists programming actions. To carry out an action, you select it from this list.
- **Status window —** which identifies download and upload files.

When you finish programming MCUs, the main screen reappears.

**Syntax:**

**PROGRAM**

**Example:**

| | |
|---|---|
| **>PROGRAM** | Start the EPROM programmer. |

# PRTA Set Port A Output Latches PRTA

The **PRTA** command is an alternative form of the **PORTA** command.

**NOTE**

If the JICS circuit board is connected , the system sends the *n* parameter value of this command to the board.

**Syntax:**

**PRTA** ***n***

where:

*n* The new value for the port A output register latches.

# PRTB Set Port B Output Latches PRTB

The **PRTB** command is an alternative form of the **PORTB** command.

**NOTES**

If the JICS circuit board is connected , the system sends the *n* parameter value of this command to the board.

Port B consists of six bits; the **PRTB** command ignores the upper two bits of larger values.

**Syntax:**

**PRTB** ***n***

where:

*n* The new value for the port B output register latches.

# QUIET Suspend Window Updates QUIET

The **QUIET** command stops updates of values in the memory, variable, chip visualization, and CPU windows during code execution (that is, while a **GO** or **GOTIL** command is under way).

The default ICS05J1A arrangement is for values in these windows to update as code executes. But entering the **QUIET** command before entering a **GO** or **GOTIL** command directs the system to replace window values with dashes during code execution, significantly increasing code execution speed.

When code execution ends, the final values reappear in the windows. Each successive time you execute code, dashes again replace the window values.

Entering the **QUIET** command a second time restores window-value updates as code executes.

**Syntax:**

**QUIET**

**Example:**

>**QUIET** Suspend window updates during code execution.

# QUIT Exit Program QUIT

The **QUIT** command terminates the host session and returns to DOS or RAPID. (The **EXIT** command is identical.)

**Syntax:**

**QUIT**

**Example:**

>**QUIT** Return to DOS or RAPID.

# R Set Up System Registers R

The **R** command starts interactive setup of such system registers as the I/O, timer, and COP.

Entering this command opens the register files window, which initially shows a list of HC05JX register files (which are sold separately). Selecting a file brings up a display of values and significance for each bit of the register.

An alternate way to bring up the register files window is to press the **F7** key. To make this window disappear, press **F10** or **Esc**.

**NOTE**

The **R** command pertains to P&E register files, sold separately.

**Syntax:**

**R**

**Example:**

| | |
|---|---|
| **>R** | Start interactive system register setup. |

# REG Show Registers REG

The **REG** command displays the contents of the CPU registers in the debug window.

**Syntax:**

**REG**

**Example:**

**>REG** Display the contents of the CPU registers.

# REM — Add Comment to Script File — REM

The **REM** command marks a script file display comment. When you execute the script file, the comment appears in the debug window. If you put several **REM** commands in a script file, the different comments appear in the debug window as file execution proceeds.

**NOTE**

Do not enter the **REM** or **WAIT** command in the debug window. Unlike other ICS05J1A commands, **REM** and **WAIT** are valid only inside script files.

**Syntax:**

**REM** ***text***

where:

| | |
|---|---|
| *text* | The display comment. You need not enclose *text* in quotes; pressing **Enter** terminates *text*. |

**Example:**

| | |
|---|---|
| **>REM Program executing** | Display message `Program executing` during script file execution. |

# RESET Simulate Processor Reset RESET

The **RESET** command resets the emulation MCU and sets the program counter to the contents of the reset vector. This command does *not* start execution of user code. (To reset and execute code, use the **RESETGO** command.)

**Syntax:**

**RESET**

**Example:**

>**RESET** Reset the MCU.

# RESET_COUNTS Reset Instruction Counters RESET_COUNTS

The **RESET_COUNTS** command resets (that is, assigns the value 0 to) all the instruction counters.

To set an instruction counter, use the **COUNTER** command. To remove an instruction counter, use the **NOCNT** command.

**Syntax:**

**RESET_COUNTS**

**Example:**

>**RESET_COUNTS** Assign value 0 to all instruction counters.

# RESETGO Reset and Restart MCU RESETGO

The **RESETGO** command resets the emulation MCU, sets the program counter (PC) to the contents of the reset vector, then starts execution from that address.

**Syntax:**

**RESETGO**

**Example:**

**>RESETGO** Reset the MCU and go.

# RUN Begin Program Execution RUN

The **RUN** command starts execution of code in the simulator, beginning at the current program counter (PC) address, or at an optional *startaddr* address. Execution continues until it encounters a breakpoint, until it encounters an error, or until you press a key.

If the **RUN** command includes both *startaddr* and *endaddr* values, code execution begins at the *startaddr* address and ends at the *endaddr* address.

Like the **GO** command, the **RUN** command is an optional forms of the **G** command.

**Syntax:**

**RUN [*startaddr* [*endaddr*]]**

where:

| | |
|---|---|
| *startaddr* | Optional execution starting address. If the command does not have a startaddr value, execution begins at the current PC value. |
| *endaddr* | Optional execution ending address. |

**Examples:**

| | |
|---|---|
| **>RUN** | Begin code execution at the current PC value. |
| **>RUN 346** | Begin code execution at address 346. |
| **>RUN 400 471** | Begin code execution at address 400. End code execution just before the instruction at address 471. |

# SCRIPT Execute Script File SCRIPT

The **SCRIPT** command executes a script file: a file that contains a sequence of debug commands. Executing the script file has the same effect as executing the individual commands, one after another. (The **MACRO** command is identical.)

Entering this command without a *filename* value brings up a list of script (.MAC) files in the current directory. You can select a file to be executed directly from this list.

Note that a script file can contain the **SCRIPT** or **MACRO** command; in this way, you can nest script files as many as 16 levels deep.

Also note that the most common use of the **REM** and **WAIT** commands is within script files. The **REM** command displays your comment while the script file executes; the **WAIT** command establishes a pause between the execution of the script-file commands.

If you give a script file the filename `STARTUP.05J`, startup routines run the script file each time you start the ICS05J1A.

**Syntax:**

**SCRIPT [*filename*]**

where:

| | |
|---|---|
| *filename* | The name of a script file. The *filename* value need not include the .MAC extension. The *filename* value can be a pathname that includes an asterisk (*) wildcard character; if so, the software displays a list of script files, for your selection. |

**Examples:**

| | |
|---|---|
| **>SCRIPT INIT.MAC** | Execute commands in file INIT.MAC. |
| **>SCRIPT *** | Display names of all .MAC files (then execute the selected file). |
| **>SCRIPT A:*** | Display names of all .MAC files in drive A (then execute the selected file). |
| **>SCRIPT** | Display names of all .MAC files in the current directory, then execute the selected file). |

# SHELL — Shell to DOS — SHELL

The **SHELL** command shells to DOS in the host computer. If the command includes an optional DOS *command*, that DOS command executes immediately.

To return to the ICS05J1A from DOS, enter **EXIT** at the DOS prompt.

**NOTE**

Be aware of memory limitations when shelling to DOS. Entering the SHELL command leaves ICS05J1A running. This means that shelling to DOS and executing a DOS command could require more memory than you have. To check the amount of available memory, enter the ICS05J1A **SYSINFO** command or the DOS **MEM** command.

**Syntax:**

**SHELL [*command*]**

where:

*command* Optional DOS command to be executed immediately.

**Example:**

**>SHELL** Access the DOS shell. To return to the emulator session, type **EXIT** at the DOS prompt.

# SHOW Display Memory SHOW

The **SHOW** command displays (in the memory window) the contents of 32 controller memory locations, beginning at the specified address. If a log file is open, this command also writes the first 16 values to the log file. (The **MD** command is identical.)

**Syntax:**

**SHOW *address***

where:

| | |
|---|---|
| *address* | The starting memory address for display in the memory window. |

**Example:**

| | |
|---|---|
| >**SHOW 300** | Display the contents of 32 bytes of memory beginning at address 300. |

# SHOWCHIP Open Chip Window SHOWCHIP

The **SHOWCHIP** command brings up the chip visualization window. This window, which overlays the variables window, depicts the MCU being emulated. Arrows indicate whether each pin is an input or output; values being simulated on pins appear beside the arrows.

Use the **HIDECHIP** command to close the chip visualization window, restoring the variables window.

**NOTE**

The chip visualization window disables zooming. Neither the **ZOOM** command nor the **Alt-Z** hotkey combination has any effect while ICS05J1A displays the chip visualization window.

**Syntax:**

**SHOWCHIP**

**Example:**

>**SHOWCHIP** Open the chip visualization window.

# SHOWCODE Display Code SHOWCODE

The **SHOWCODE** command displays the source code around the specified address, without changing the value of the program counter (PC). The code window shows the preceding line of code, the specified line of code, and the next nine lines of code.

**Syntax:**

**SHOWCODE *address***

where:

| | |
|---|---|
| *address* | The memory address for code-window display. |

**Example:**

| | |
|---|---|
| **>SHOWCODE 400** | Show code around location $400 (that is, from $3FF through $409). |

# SHOWPC Display Code at PC SHOWPC

The **SHOWPC** command displays source code around the address in the program counter (PC). The code window shows the preceding line of code, the line of code the PC specifies, and the next nine lines of code. (This command often is useful immediately after the **SHOWCODE** command.)

**Syntax:**

**SHOWPC**

**Example:**

| | |
|---|---|
| >**SHOWPC** | Show code around the PC address value (that is, from PC-1 through PC+9). |

# SNAPSHOT Save Screen SNAPSHOT

The **SNAPSHOT** command saves a copy of the main screen to the open log file. (A log file must be open, or this command has no effect.)

**NOTE**

The main screen includes certain extended ASCII characters. When you subsequently view a screen snapshot, a standard ASCII editor will display a few characters do not match the original screen. But all values and other important simulation information will be in the snapshot, just as they were on the screen.

**Syntax:**

**SNAPSHOT**

**Example:**

**>SNAPSHOT** Capture screen, save to the log file.

# SOURCE Toggle Code Display SOURCE

The **SOURCE** command toggles the code window display between source code and disassembled code.

If you enter this command when the code window displays disassembled code, when a map file is loaded, and when the PC points to a memory area covered by the map file, the window display changes to source code. (The title of the window changes to SOURCE.)

**NOTE**

If you alter code space defined in the source file(via the **ASM**, **MEM**, or **MM** command), only the disassembled-code display reflects your changes. (In other words, the source-code display no longer will be correct.) To incorporate your changes into the source-code display, you must use RAPID and CASM05J to re-assemble the code.

**Syntax:**

**SOURCE**

**Example:**

>**SOURCE** Toggle the display in the code window.

# SP Set SP Value SP

The **SP** command assigns the specified value to the stack pointer (SP).

**Syntax:**

**SP** ***n***

where:

*n* The new SP value, *C0* through *FF*.

**Example:**

**>SP $E0** Set the SP value to *$E0*.

# SS Source Step SS

The **SS** command steps through one or a specified number of *source-code* instructions, beginning at the current program counter (PC) address value. This makes the **SS** command useful to high-level-language compilers, for stepping through compiler source code instead of the assembly instructions.

If you enter this command without any parameter value, it steps through one source-level instruction. If you enter the **SS** command with an *n* value, the command steps through that many source-level instructions.

**Syntax:**

**SS [*n* ]**

where:

*n* Optional number of instructions to step through.

**Examples:**

| | |
|---|---|
| **>SS** | Step through the instruction at the PC address value. |
| **>SS 8** | Step through eight instructions, starting at the PC address value. |

# ST Single Step (Trace) ST

The **ST** command steps through one or a specified number of assembly instructions, beginning at the current program counter (PC) address value. Then this command halts. If you enter this command without any parameter value, it steps through one instruction. If you enter the **ST** command with an *n* value, the command steps through that many instructions.

(The **ST**, **STEP**, and **T** commands are identical.)

**Syntax:**

**ST [*n* ]**

where:

*n* The *hexadecimal* number of instructions to be executed.

**Examples:**

| | |
|---|---|
| **>ST** | Execute the assembly instruction at the PC address value. |
| **>ST 2** | Execute two assembly instructions, starting at the PC address value. |

# STACK Display Stack STACK

The **STACK** command depicts the ICS05J1A stack. Entering this command brings up the stack window, which shows the stack pointer (SP) value and the top 12 raw-byte values of the stack.

During an interrupt, the stack window also shows the interrupt stack: the top five values of the stack, plus the values of the condition code register (CCR), accumulator (A), and index (X) register. This information indicates the restored state of the stack upon the return from the interrupt.

During execution of a subroutine, the stack window also shows the subroutine stack, which indicates the restored state of the stack upon the return from the subroutine.

**NOTE**

M68HC05 MCUs store information in the stack (1) during an interrupt or (2) during execution of a subroutine. The stack window shows both these possible interpretations of stack data. You must know whether program execution is in an interrupt or in a subroutine, to know which stack data interpretation is valid.

An alternate way to bring up the stack window is to press the **F2** key. To make the stack window disappear, press the **F10** key.

**Syntax:**

**STACK**

**Example:**

>**STACK** Open the stack window.

# STEP Single Step (Trace) STEP

The **STEP** command steps through one or a specified number of assembly instructions, beginning at the current program counter (PC) address value. Then this command halts. If you enter this command without any parameter value, it steps through one instruction. If you enter the **STEP** command with an *n* value, the command steps through that many instructions.

(The **ST**, **STEP**, and **T** commands are identical.)

**Syntax:**

**STEP [*n* ]**

where:

*n* The *hexadecimal* number of instructions to be executed.

**Examples:**

| | |
|---|---|
| **>STEP** | Execute the assembly instruction at the PC address value. |
| **>STEP 2** | Execute two assembly instructions, starting at the PC address value. |

# STEPFOR Step Forever STEPFOR

The **STEPFOR** command continuously executes instructions, beginning at the current program counter (PC) address value. Execution stops when it arrives at a breakpoint, when it encounters an error, or when you press a key.

**Syntax:**

**STEPFOR**

**Example:**

>**STEPFOR** Execute instructions continuously until the user presses a key.

# STEPTIL Step Until Location STEPTIL

The **STEPTIL** command continuously executes instructions, from the current program counter (PC) address value until the PC value reaches the specified address.

**Syntax:**

**STEPTIL *address***

where:

*address* Execution stop address. This must be an instruction address.

**Example:**

>**STEPTIL 0400** Execute instructions continuously until the PC value is 0400.

# SYMBOL Add Symbol SYMBOL

The **SYMBOL** command creates a new symbol, which you can use anywhere, in place of the symbol value. If you enter this command without any parameter values, the software displays the list of current user-defined symbols.

This command has the same effect as an EQU statement in the assembler.

**Syntax:**

**SYMBOL [*label value*]**

where:

*label* The ASCII-character string label of the new symbol.

*value* The value of the new symbol.

**Examples:**

| | |
|---|---|
| **>SYMBOL** | Show the current user-defined symbols. |
| **>SYMBOL timer_control $08** | Define new symbol `timer_control`, with value $08. (Subsequently, to modify location 08, you could enter the command **MM timer_control**.) |

# SYSINFO Show System Information SYSINFO

The **SYSINFO** command shows the amount of system memory available to the simulator, including the largest memory block available.

**Syntax:**

**SYSINFO**

**Example:**

>**SYSINFO** Show system information.

```
Total memory available: 187488  Largest free block: 187488
```

# T Single Step (Trace) T

The **T** command steps through one or a specified number of assembly instructions, beginning at the current program counter (PC) address value. Then this command halts. If you enter this command without any parameter value, it steps through one instruction. If you enter the **T** command with an *n* value, the command steps through that many instructions.

(The **ST**, **STEP**, and **T** commands are identical.)

**Syntax:**

**T [*n* ]**

where:

*n* The *hexadecimal* number of instructions to be executed.

**Examples:**

| | |
|---|---|
| **>T** | Execute the assembly instruction at the PC address value. |
| **>T 2** | Execute two assembly instructions, starting at the PC address value. |

# UPLOAD_SREC Upload S-Record UPLOAD_SREC

The **UPLOAD_SREC** command uploads the contents of the specified memory block (range), in .S19 format, displaying the contents in the debug window.

**NOTE**

If you enter the **UPLOAD_SREC** command, the memory contents scroll through the debug window too rapidly to view. Accordingly, you first should enter the **LOGFILE** command, which records the contents in a log.

**Syntax:**

**UPLOAD_SREC *startrange endrange***

where:

| | |
|---|---|
| *startrange* | Beginning address of the memory block (range). |
| *endrange* | Ending address of the memory block (range). |

**Example:**

| | |
|---|---|
| **>UPLOAD_SREC 300 7FF** | Upload the 300—7FF memory block in .S19 format. |

# VAR Display Variable VAR

The **VAR** command specifies an address for monitoring, adding the address and its current contents to the variables window. As the value at the address changes, the variables window updates the value.

The byte, word, or string variant determines the display format:

- Byte (**.B**): hexadecimal and binary (the default)
- Word (**.W**): hexadecimal and decimal
- String (**.S**): ASCII characters

The optional *n* parameter specifies the number of string characters to be displayed; the default value is 12. (The *n* parameter has no effect for byte or word values.)

**Syntax:**

**VAR[.B|.W|.S]** ***address*** **[*n*]**

where:

*address* The address of the memory variable.

*n* Optional number of characters for a string variable; default value is 12, does not apply to byte or word variables.

**Examples:**

| | |
|---|---|
| >**VAR C0** | Show byte value of address C0 (hexadecimal and binary). |
| >**VAR.B D4** | Show byte value of address D4 (hexadecimal and binary). |
| >**VAR.W E0** | Show word value of address E0 (hexadecimal and decimal). |
| >**VAR.S C0 5** | Show the five-character ASCII string at address C0. |

# VER Display Version VER

The **VER** command displays the version and date of ICS05J1A. (**VERSION** is an alternate form of this command.)

**Syntax:**

**VER**

**Example:**

**>VER** Display ICS05J1A version.

# VERSION Display Version VERSION

The **VERSION** command displays the version and date of the ICS05J1A. (**VER** is an alternate form of this command.)

**Syntax:**

**VERSION**

**Example:**

>**VERSION** Display ICS05J1A version.

# WAIT Wait N Cycles WAIT

The **WAIT** command delays simulator command execution by the specified number of cycles.

This command works in script files to control when inputs come into the simulator. When execution encounters a WAIT command, it passes control back to the keyboard. That is, script-file execution waits for you to enter a command such as **GO** or **STEP**, which starts MCU cycles. As soon as the number of cycles equals the n value of the **WAIT** command, the simulator resumes executing commands of the script file.

**NOTES**

Do not enter the **REM** or **WAIT** command in the debug window. Unlike other ICS05J1A commands, **REM** and **WAIT** are valid only inside script files.

The **WAIT** command must not be the last command of a script file.

**Syntax:**

**WAIT *n***

where:

*n* The *hexadecimal* number of cycles to pause.

**Example:**

**>WAIT A** Delay command execution for 10 MCU cycles.

# WHEREIS Display Symbol Value WHEREIS

The **WHEREIS** command displays a symbol or an address. If you enter this command with a *symbol* argument, the result is the address of the symbol. If you enter this command with an *address* argument, the result is the symbol at that address (provided that such a symbol exists).

**Syntax:**

**WHEREIS *symbol* | *address***

where:

| | |
|---|---|
| *symbol* | A symbol listed in the symbol table. |
| *address* | Address for which a symbol is defined. |

**Examples:**

| | |
|---|---|
| **>WHEREIS START** | Display the symbol START and its value. |
| **>WHEREIS 0300** | Display the value *0300* and its symbol, if any. |

# X Set X Register Value X

The **X** command sets the index register (X) to the specified value. (The **X** command is identical to the **XREG** command.)

**Syntax:**

**X *value***

where:

*value* The new value for the X register.

**Example:**

>**X 05** Set the index register value to 05.

# XREG Set X Register Value XREG

The **XREG** command sets the index register (X) to the specified value. (The **XREG** command is identical to the **X** command.)

**Syntax:**

**XREG *value***

where:

*value* The new value for the X register.

**Example:**

>**XREG F0** Set the index register value to F0.

# Z Set/Clear Z Bit Z

The **Z** command sets or clears (that is, assigns *0* or *1* to) the Z bit of the condition code register (CCR).

**NOTE**

The CCR bit designators are at the lower right of the CPU window. The CCR pattern is 111HINZC (H is half-carry, I is IRQ interrupt mask, N is negative, Z is zero, and C is carry). A letter in these designators means that the corresponding bit of the CCR is set; a period means that the corresponding bit is clear.

**Syntax:**

```
Z 0|1
```

**Examples:**

| | |
|---|---|
| >**Z 0** | Clear the Z bit of the CCR. |
| >**Z 1** | Set the Z bit of the CCR. |

# ZOOM Resize Source Window ZOOM

The **ZOOM** command toggles the size of the source window between normal and enlarged.

An alternative way to toggle the window size is to press the **Alt-Z** keys.

**NOTE**

The chip visualization window disables zooming. Neither the **ZOOM** command nor the **Alt-Z** hotkey combination has any effect while ICS05J1A displays the chip visualization window.

**Syntax:**

**ZOOM**

**Example:**

>**ZOOM** Resize the source window.

# CHAPTER 4
# SUPPORT INFORMATION

This chapter consists of support information for the M68HC705J1A In-Circuit Simulator Kit. Paragraph 4.1 is a functional description of the kit. Paragraph 4.2 is a quick-start troubleshooting guide; paragraph 4.3 is a programming troubleshooting guide. Paragraph 4.4 contains the kit logic diagram (Figure 4-1).

## 4.1 FUNCTIONAL DESCRIPTION

The M68HC705JICS circuit board contains two separate sections: one for J1A emulation and the other for 705J1A programming (including VPP generation).

### 4.1.1 Emulation

At the heart of the emulation section is an M68HSC705C8A MCU. This MCU provides all necessary I/O information, so that the host computer can carry out emulation. That is, the computer performs all simulation functions, except maintaining port values. The C8A MCU runs a program from its internal EPROM to generate appropriate port values and communicate with the host computer.

When the computer, acting as the simulator, needs port data, the computer sends a data request through the serial port to the C8A MCU. The C8A MCU sends the data back through the serial port to the host computer. This arrangement means that the simulator, normally an I/O-restrictive program, can interface to the real world.

The circuit board 7.38-megahertz crystal provides a clock signal for the C8A MCU; the clock runs the MCU at a 3.69-megahertz bus rate. Note that simulation speed is less than the bus rate, as the host computer is the simulator.

Note that the C8A MCU (unlike the J1A MCU) does *not* have programmable pull-downs. Accordingly, the JICS circuit board has external pull-down resistors, selectable via jumper headers J3 through J16. Remove the fabricated jumpers from any of these headers to disable the corresponding pull-downs.

### 4.1.2 Programming

Programming software in the host computer controls the JICS circuit board programming sockets (U6 and U7). The C8A MCU communicates with the computer via its serial interface, and handles all programming tasks with its I/O pins.

The circuit board's MC34063 MCU generates the programming voltage (16.5 volts). Switch S2 controls this voltage at the programming sockets. (The software prompts you to turn switches S1 and S2 on and off at the appropriate times.)

## 4.2 QUICK-START TROUBLESHOOTING GUIDE

In case of problems, when you quick-start your kit, follow these steps:

**Step 1.** If the M68HSC705C8A MCU (at circuit board location U4) has a window, make sure that a black opaque label covers the window.

**Step 2.** Reboot your computer without loading system startup files. This eliminates TSRs, mode commands, and similar routines that could interfere with kit operation.

**Step 3.** Make sure that you have the correct serial cable.

**Step 4.** Disconnect the circuit board from the wall transformer, then measure the transformer output. This output should be greater than 9 volts dc, but less than 15 volts. If voltage is within this range, proceed to step 5. If voltage is outside this range, call Global Data Specialists (800-451-3464 or 602-437-4331) for a replacement.

**Step 5.** Reconnect the transformer to the circuit board to measure the voltage at pin 3 of the 78T05 regulator (location U1.) With switch S1 turned ON, the voltage at U1 pin 3 should be 5 volts, ± 0.5 volts. If the voltage is less than 4.5 volts, make sure that your power supply can provide sufficient current; if *not*, correct the condition. If there is sufficient current but the U1, pin 3, voltage still is less than 4.5 volts, remove the C8A MCU. If the voltage at U1 pin 3 climbs to the correct range, the C8A MCU is defective; call Global Data Specialists (800-451-3464 or 602-437-4331) to arrange for a replacement. If the voltage remains below 4.5 volts even with the C8A MCU removed, the circuit board is defective; call Global Data Specialists for a replacement.

**NOTE**

To replace the C8A MCU, you must use a *programmed* replacement MCU; substituting just any C8A MCU is not satisfactory. If a programmed replacement is not available, you may be able to program a replacement yourself, by downloading file ICS05J.S19 from Motorola's BBS at 512-891-3733 or Motorola's web site at http://design-net.com/csic/CSIC_home.html.

**Step 6.** If you measure 5 volts at pin 3 of the 78T05 regulator when the C8A MCU is installed, measure the voltage between VDD (pin 40) and VSS (pin 20) of the C8A MCU. If the level is not also 5 volts, check for a bent pin or other structural problem with the socket or the board trace. If you cannot find a structural problem, call Global Data Specialists (800-451-3464 or 602-437-4331) to arrange for a board replacement.

**Step 7.** If there *are* 5 volts between C8A pins 40 and 20, use an oscilloscope to check the output of pin 38 of the MCU. Set the oscilloscope to 0.5 microseconds per division. You should observe one cycle per division, which corresponds to a signal of about 7.3 megahertz. If you do not get this result, any of these problems could exist:

- bad crystal (location Y1)
- bad resistor (location R16)
- bad capacitor (location C13 or C17)
- bad C8A MCU (location U4)
- bad socket (location U4)
- broken trace on the circuit board
- cold solder joint on the circuit board

**Step 8.** If you have not yet found the problem, measure two signals when you enter the **POD** command: pin 3 of the RS-232 serial connector (location P2) and pin 29 of the C8A MCU (location U4). The P2, pin 3, signal should be at RS-232 level (-5 to +12 volts); the U4, pin 29, signal should be at TTL level (0 to 5 volts). If signals appear on P2, pin3, but not on U4, pin 29, check the circuit board components near location Q3. If signals do not appear on P2, pin 3, check the serial cable or the selected communications port.

**Step 9.** A final possible cause of problems is that LED1 is defective. To check this possibility, remove the C8A MCU from location U4, then short socket pin 4 to ground. If LED1 does not light, it is defective or installed backwards.

**Step 10.** If you still cannot complete step 7 of the Chapter 1 quick-start instructions (that is, if LED1 still does not flash) consult a field application engineer from you Motorola distributor or sales office.

**Step 11.** If you cannot compete step 8 of the Chapter 1 quick-start instructions (that is, if LED1 continues to flash, check that connector pin PA0 goes high when you push switch S4. If the pin does not go high, ohm out resistor R10 and switch S4. Consult Global Data Specialists (800-451-3464 or 602-437-4331), as appropriate.

## 4.3 PROGRAMMING TROUBLESHOOTING GUIDE

Should you run into problems when programming a 705J1A MCU (but you can perform other simulator functions) follow these steps:

**Step A.** Make sure that both programming sockets (U6 and U7) are empty. Verify that the voltage on pin 19 of the programming sockets is 16.5 volts. If voltage is less than 16 volts or greater than 17 volts, the socket may be bad or there may be a bad circuit-board trace. To find such a problem, go through quick-start troubleshooting, per paragraph 4.2.

**Step B.** If pin 19 of either programming socket does not have 16.5 volts, but is in the 16- to 17-volt range, calibrate VPP. To do so, turn switch S1 ON, turn switch S2 ON, then measure the signal on pin 19 of socket U6. Adjust potentiometer VR1 until your meter reads 16.5 volts. Then turn switches S2 and S1 OFF. (This completes VPP calibration.).

**Step C.** Type **`PROGRAM`** and follow the instructions that appear on the programming screen. Before you insert the 705J1A MCU into a programming socket, make sure that the MCU erase window is covered.

**Step D.** Choose the `BLANK CHECK` programming option. You should see activity on pin 1 of socket U6 or U7. If there is no activity, there is a circuit-board or M68HSC705C8A MCU problem; call Global Data Specialists (800-451-3464 or 602-437-4331), to arrange for a replacement.

**Step E.** If the blank check result is `PART NOT BLANK`, erase (or replace) the 705J1A MCU.

**Step F.** If there is activity on pin 1 of socket U6 or U7, choose the `PROGRAM` option. Use an oscilloscope to measure VPP as the 705J1A MCU is being programmed; VPP should not drop below 16 volts. Should VPP drop below 16 volts, the 705J1A MCU or the MC34063 MCU is defective; replace these MCUs or call Global Data Specialists (800-451-3464 or 602-437-4331).

## 4.4. LOGIC DIAGRAM

Figure 4-1 is the JICS logic diagram.

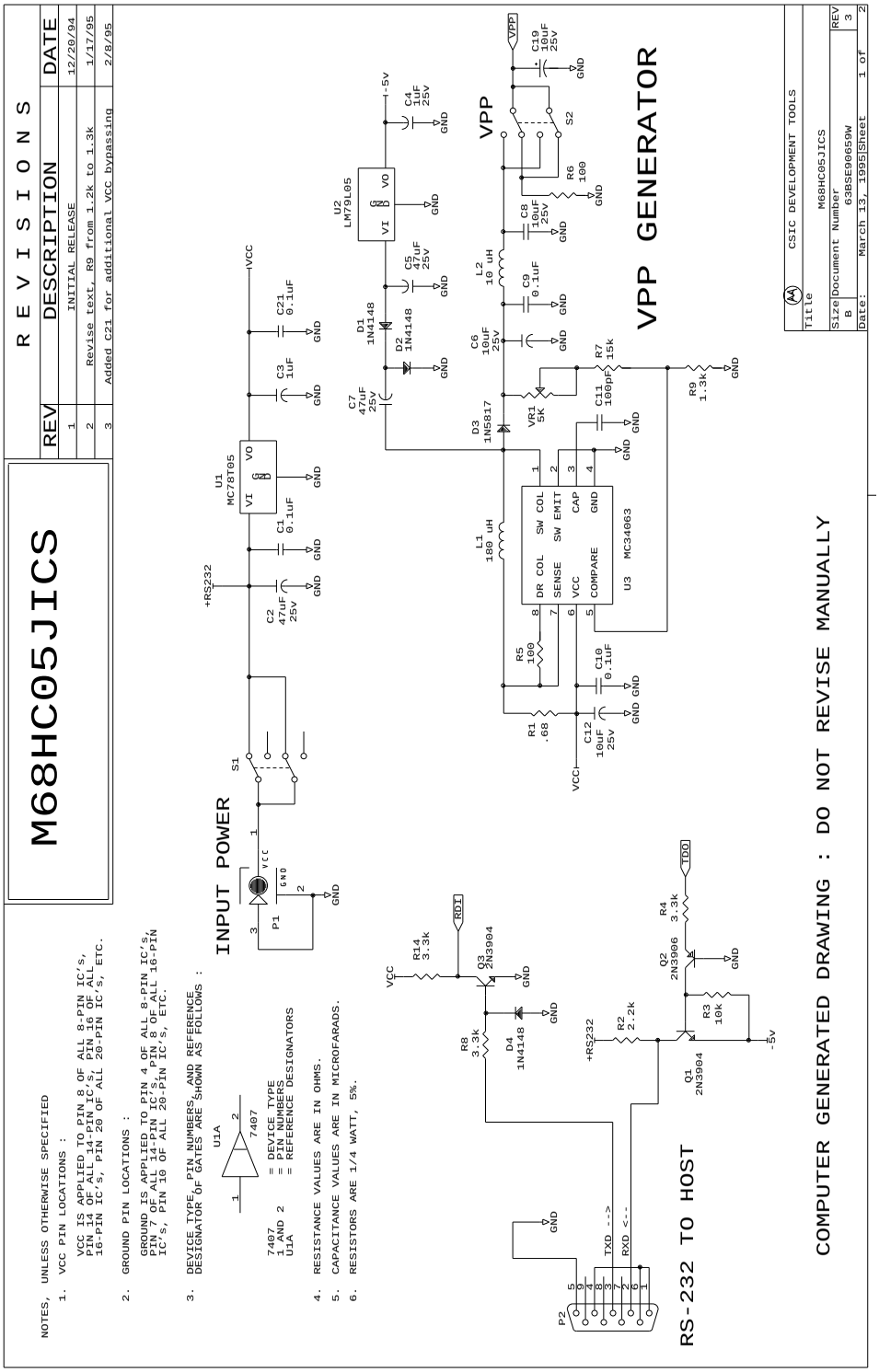

M68HC05JICS
REVISIONS
REV DESCRIPTION DATE
1 INITIAL RELEASE 12/20/94
2 Revise text, R9 from 1.2k to 1.3k 1/17/95
3 Added C21 for additional VCC bypassing 2/8/95
NOTES, UNLESS OTHERWISE SPECIFIED
1. VCC PIN LOCATIONS :
VCC IS APPLIED TO PIN 8 OF ALL 8-PIN IC's, PIN 14 OF ALL 14-PIN IC's, PIN 16 OF ALL 16-PIN IC's, PIN 20 OF ALL 20-PIN IC's, ETC.
2. GROUND PIN LOCATIONS :
GROUND IS APPLIED TO PIN 4 OF ALL 8-PIN IC's, PIN 7 OF ALL 14-PIN IC's, PIN 8 OF ALL 16-PIN IC's, PIN 10 OF ALL 20-PIN IC's, ETC.
3. DEVICE TYPE, PIN NUMBERS, AND REFERENCE DESIGNATOR OF GATES ARE SHOWN AS FOLLOWS :
7407 = DEVICE TYPE
1 AND 2 = PIN NUMBERS
U1A = REFERENCE DESIGNATORS
4. RESISTANCE VALUES ARE IN OHMS.
5. CAPACITANCE VALUES ARE IN MICROFARADS.
6. RESISTORS ARE 1/4 WATT, 5%.
INPUT POWER
VPP
VPP GENERATOR
RS-232 TO HOST
COMPUTER GENERATED DRAWING : DO NOT REVISE MANUALLY
CSIC DEVELOPMENT TOOLS
M68HC05JICS
63BSE906659W
March 13, 1995

**Figure 4-1. JICS Logic Diagram**

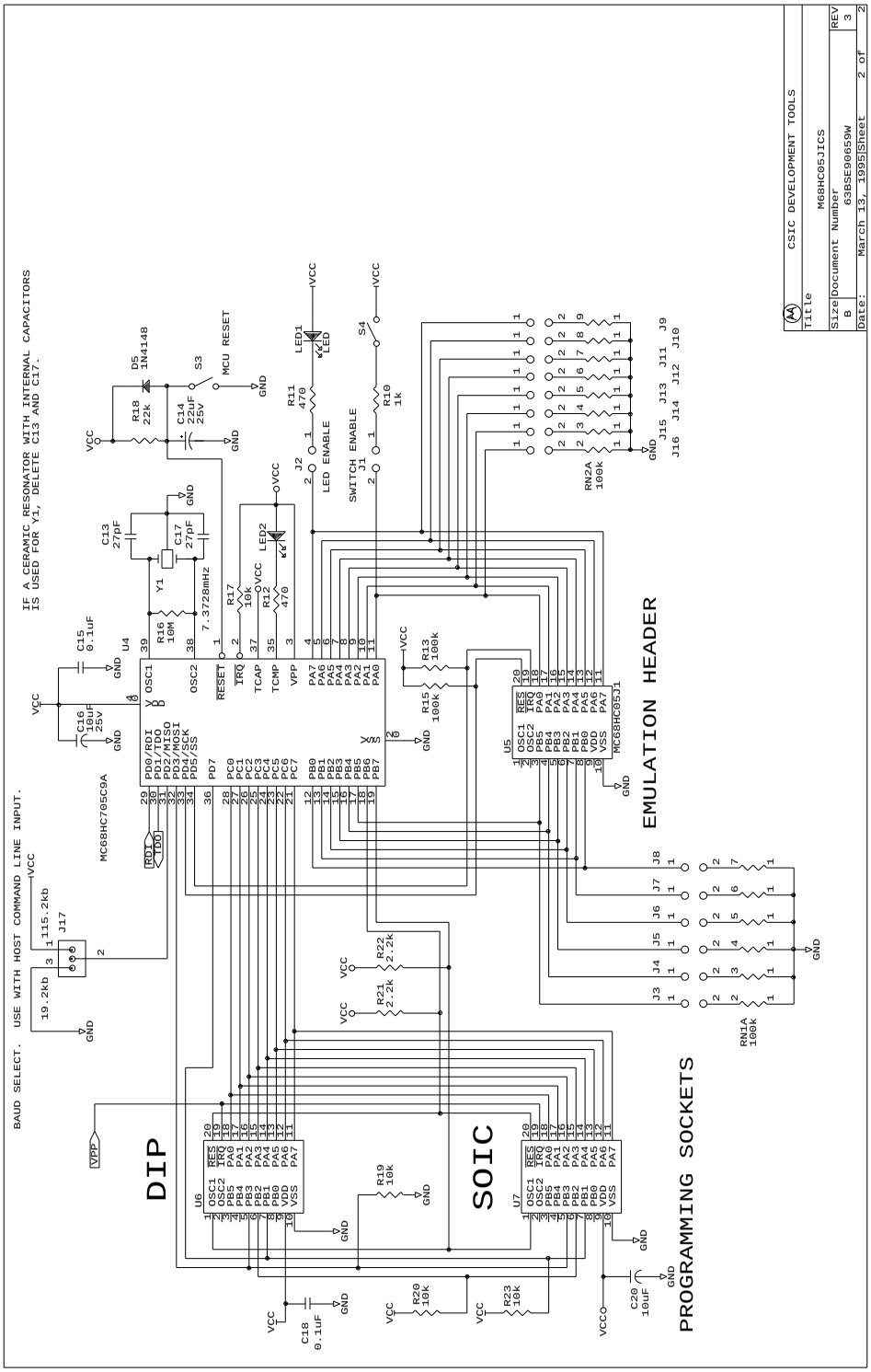



**Figure 4-1. JICS Logic Diagram (continued)**

# INDEX

A command: 3-6

ACC command: 3-7

Accumulator breakpoints: 3-12, 3-62

ASM command: 2-4, 2-9, 3-8

BELL command: 3-9

BF command: 3-10

BR command: 3-11

BREAKA command: 3-12

Breakpoint window: 2-7

Breakpoints:
- accumulator: 3-12, 3-62
- index: 3-14, 3-62
- instruction: 3-11, 3-62
- stack pointer: 3-13, 3-62

BREAKSP command: 3-13

BREAKX command: 3-14

C command: 3-15

CAPTURE command: 2-10, 3-16

Capture files: 2-10

CAPTUREFILE command: 2-10, 3-17

CCR command: 3-18

CF command: 2-10, 3-19

Changing screen colors: 2-7, 2-10, 3-22

Chip visualization window: 2-7

CLEARMAP command: 3-20

CLEARSYMBOL command: 3-21

Code window: 2-3 — 2-5

COLORS command: 2-7, 2-9, 2-10, 3-22

Color selection window: 2-7

Colors, changing: 2-10

Commands, ICS05J1A: 3-1 — 3-107

COUNTER command: 2-7, 3-23

CPU window: 2-3

CY command: 1-8, 2-3, 3-24

CYCLES command: 2-3, 3-25

DASM command: 3-26

DDRA command: 3-27

DDRB command: 3-28

DUMP command: 3-29

Debug window: 2-6

Error messages and warnings: 2-11

EVAL command: 3-30

EXIT command: 1-3, 3-31

Features: 1-1, 1-2

File types: 2-1

Funtional description: 4-1, 4-2

G command: 1-8, 3-32

GO command: 1-3, 1-7, 3-33

GOMACRO command: 3-34

GOTIL command: 2-4, 3-35

GOTOCYCLE command: 3-36

H command: 3-37

Hardware: 1-3 — 1-6:
- circuit board layout: 1-3, 1-4
- confinguring jumper headers: 1-4 — 1-6

HELP command: 2-9, 3-38

Help system: 2-9

HIDECHIP command: 2-7, 3-39

History:
- buffer: 2-11
- capture: 2-11
- window: 2-7

HISTORY command: 2-7, 2-11, 3-40

HISTORYLOG command: 2-11, 3-41

Hotkeys, ICS05J1A: 2-7, 2-8

I command: 3-42

ICS05J1A:
- hotkeys: 2-7, 2-8
- starting: 2-1, 2-2
- commands: 3-1 —3-107

Index breakpoints: 3-14, 3-62

INFO command: 3-43

INPUTA command: 3-44

INPUTB command: 3-45

INPUTS command: 3-46

Instruction breakpoints: 3-11, 3-62

Instruction counter window: 2-7

INT command: 3-47

IRQ command: 3-48

Jumper headers, setting: 1-4, 1-5

LF command: 2-10, 3-49

LISTOFF command: 3-50

LISTON command: 3-51

LOAD command: 1-7, 1-8, 2-1, 3-52

LOADMAP command: 2-1, 3-53

Log files: 2-10

LOGFILE command: 2-10, 3-54

Logic diagram: 4-5, 4-6

Main screen: 2-2 — 2-7
code window: 2-3 — 2-5
CPU window: 2-3
debug window: 2-6
memory window: 2-6
secondary windows:
breakpoint: 2-7
chip visualization: 2-7
color selection: 2-7
history: 2-7
instruction counter: 2-7
register files: 2-7
stack: 2-7
variables window: 2-5
MACRO command: 3-55
MACROEND command: 3-56
MACROSTART command: 3-57
MD command: 2-6, 3-58
MEM command: 2-4, 2-9, 3-59
Memory window: 2-5, 2-6
MM command: 2-4, 2-9, 3-60
N command: 3-61
NOBR command: 3-62
NOCNT command: 2-7, 3-63
Numbers: 2-9
Operating the simulator: 2-1 — 2-11
Overview: 1-1 — 1-8
PC command: 1-7, 3-64
POD command: 3-65
PORTA command: 3-66
PORTB command: 3-67
PROGRAM command: 1-7, 3-68, 4-4

Programming:
quick start: 1-7
troubleshooting: 4-4

PRTA command: 3-69

PRTB command: 3-70

Quick start:
ICS05J1A: 1-2, 1-3
programming: 1-7
troubleshooting: 4-2, 4-3

QUIET command: 1-8, 3-71

QUIT command: 3-72

R command: 3-73

REG command: 3-74

Register files window: 2-7

REM command: 2-10, 3-75

RESET command: 2-11, 3-76

RESET_COUNTS command: 2-7, 3-77

RESETGO command: 3-78

RUN command: 3-79

Reset switch: 1-3, 1-4

Schematic diagram: 4-5, 4-6

Screen, main: 2-2 — 2-7

SCRIPT command: 3-80

Script files: 2-10

Setting jumper headers: 1-4 — 1-6

SHELL command: 3-81

SHOW command: 2-6, 3-82

SHOWCHIP command: 2-7, 3-83

SHOWCODE command: 3-84

SHOWPC command: 3-85

Simulation speed: 1-8

SNAPSHOT command: 3-86

SOURCE command: 3-87

SP command: 3-88

Specifications: 1-2

SS command: 3-89

ST command: 3-90

STACK command: 3-91

Stack pointer breakpoints: 3-13, 3-62

Stack window: 2-7

Starting ICS05J1A: 2-1, 2-2

STEP command: 3-92

STEPFOR command: 3-93

STEPTIL command: 3-94

Support information: 4-1—4-6

SYMBOL command: 3-95

SYSINFO command: 3-96

T command: 3-97

Troubleshooting:
- programming: 4-4
- quick-start: 4-2, 4-3

UPLOAD_SREC command: 3-98

VAR command: 3-99

Variables window: 2-5

VER command: 3-100

VERSION command: 3-101

Warnings and error messages: 2-11

WAIT command: 3-102

WHEREIS command: 3-103

Windows:
breakpoint: 2-7
chip visualization: 2-7
code: 2-3 — 2-5
color selection: 2-7
CPU: 2-3
debug: 2-6
history: 2-7
instruction counter: 2-7
memory: 2-6
register files: 2-7
stack: 2-7
variables: 2-5

X command: 3-104

XREG command: 3-105

Z command: 3-106

ZOOM command: 3-107